UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

TRABAJO FIN DE CARRERA

PROCESO DE OPTIMIZACIÓN DEL RENDIMIENTO DE CÓDIGOS CIENTÍFICOS PARA CÁLCULO DE ALTAS PRESTACIONES

Autor: Oscar de Bustos Martín
Tutor: Roberto San José García
Tutor: Félix García Merayo

2009

AGRADECIMIENTOS

Quisiera agradecer este trabajo fin de carrera a:

- En primer lugar, a mi mujer Lola, ya que su constancia y tenacidad ha hecho posible que pudiera sentarme y centrar todas las ideas que tenía en la cabeza.

- A mi hija Aitana, por alegrarme todos los fines de semana con su sonrisa cuando me levantaba pronto a escribirlo.

- A mi segundo hijo, todavía por nacer, y que seguro que cuando lea este proyecto estará casi en camino.

- A mis padres, cuyo esfuerzo privándose en muchas ocasiones de cosas para ellos, posibilitó que pudiera estudiar y realizar la carrera en la Facultad de Informática.

- A D. Félix García Merayo, tutor de este trabajo, y cuya asignatura de sexto de carrera (Procesamiento Vectorial y Paralelo) me introdujo en el mundo de la computación de altas prestaciones, sector en donde llevo trabajando desde hace más de 10 años.

- A Albert Trill, Director de Preventa de SGI en España y a Igor Zacharov Ingeniero de Optimización de Códigos también de SGI, que me enseñaron todo lo que había que saber sobre como optimizar un código científico.

ÍNDICE

Capítulo/Sección *Página*

ÍNDICE DE FIGURAS

1 RESUMEN

Este trabajo fin de carrera consiste en la implementación de una metodología para la optimización de códigos científicos.

Para ello, se estudiará primeramente las diferentes arquitecturas que han existido hasta la fecha para comprender la importancia del hardware en el rendimiento de un código.

Posteriormente se profundizará en la metodología centrándose sobre todo en la optimización de códigos monoprocesador, dejando la paralelización para futuras líneas de trabajo.

Este trabajo pretende ser una guía de referencia para aquellos científicos y administradores que quieran sacar un mayor rendimiento a los códigos científicos.

2 INTRODUCCIÓN

2.1 OBJETIVOS DEL PROYECTO

Durante mis años de formación en la Facultad de Informática y posteriormente en mi vida laboral trabajando como responsable del sector de Computación de Altas Prestaciones, o más comúnmente conocido en inglés como HPC (*High Performance Computing*), en Silicon Graphics y Bull, y de Linux en Novell, he podido comprobar cómo la forma de escribir código de una forma u otra puede afectar enormemente al rendimiento posterior de ejecución.

Por ello, los objetivos que se plantean en este trabajo de fin de carrera son los siguientes:

- Analizar las diferentes arquitecturas de computadores que han existido hasta la actualidad.
- Analizar como las características de cada arquitectura puede impactar en el rendimiento de los códigos que se ejecutan sobre esos sistemas.
- Implementar una metodología para el proceso de optimización de códigos que permita evaluar y optimizar el rendimiento de dichos códigos sobre sistemas de computación.
- Mostrar ejemplos prácticos en los cuales se implemente las técnicas de esta metodología.
- Analizar futuras opciones en computación como la aplicación de las GPU en HPC.

2.2 CONTEXTO DEL PROYECTO

Desde la aparición de los sistemas métricos en la antigüedad, los pensadores y científicos han tratado de automatizar los cálculos.

En todos estos avances que se han ido produciendo a lo largo de la historia, el objetivo final era el mismo: tratar de mejorar las prestaciones de los cálculos que se realizaban sobre ellos, bien mejorando el número de operaciones que se podían realizar o bien reduciendo el tiempo que se tardaba en realizar dichos cálculos.

Por ello, el empleo de técnicas de optimización de cálculo ha sido siempre un reto para los científicos.

En la actualidad, la computación es uno de los campos en los cuales se invierte más dinero y recursos dentro del sector de las tecnologías de la información.

Es por ello que este proyecto fin de carrera tiene como objeto implantar una metodología que permita a los programadores actuales mejorar la eficiencia y el rendimiento de los códigos para aumentar la productividad de sus sistemas.

Durante la primera parte de este trabajo fin de carrera se verán las diferentes arquitecturas y lenguajes de programación que han existido y que existen en la actualidad, puesto que son la base para diseñar metodologías de programación que permitan la mejora del rendimiento de códigos.

Durante la segunda parte de este trabajo fin de carrera, se diseñará una metodología que permita al lector mejorar con técnicas sencillas y complejas el rendimiento de sus códigos. Para ello, se va a mostrar diferentes técnicas de optimización de códigos recogidas de la experiencia propia trabajando en este campo, así como de diferentes cursos que he dado sobre esta materia en los últimos diez años en diversas Universidades y Centros de Investigación [De Bustos, 2000-2006], como otras fuentes: [Cortesi and Fier, 1998], [Gerber, Bik, Smith and Tian, 2006], [Koren, 2006], [Snyder, 2000], [Trill, 2000], [Vogelsang, 2005] y [Zacharov, 2001].

Este trabajo fin de carrera es de lectura obligada para aquellos científicos, usuarios y administradores que quieran aprovechar las técnicas descritas en él para mejorar el rendimiento y la eficiencia en los centros de cálculo y demostrar que implantar un código eficiente no tiene que ser una tarea compleja y que todas estas técnicas descritas pueden ser aplicadas en la realidad.

2.3 IMPORTANCIA DEL PROYECTO

Actualmente el modelo de investigación implantado en la mayoría de los países occidentales obliga a competir a los científicos unos frente a otros. Aquellos que más publicaciones y trabajos de investigación saquen al mercado recibirán más méritos para conseguir, bien futuras plazas de investigador o bien recibir más subvenciones por parte de las administraciones públicas.

Para realizar sus investigaciones, la mayoría de los científicos actualmente necesitan sistemas informáticos para plasmar sus modelos teóricos.

En la mayoría de los casos el mero hecho de disponer de grandes computadores para poder utilizar sus recursos no siempre es suficiente. Que duda cabe, que cuantos más recursos informáticos tiene a su disposición el científico de hoy en día, mayor complejidad es capaz de abordar en sus problemas y mejores son los resultados que

puede obtener. Sin embargo, la complejidad de cálculos que son capaces de realizar hoy en día los científicos desborda muchas veces los recursos que se tienen disponibles y lo que necesitan es tratar de optimizar los códigos para emplear mejor dichos recursos.

No obstante, siempre hay una balanza que se debe valorar por encima de todo: no se puede perder mucho tiempo optimizando los códigos si ello implica que perdamos mucho tiempo en publicar nuestros resultados.

Uno de los mayores problemas que tienen a día de hoy los científicos, y sobre todo en España, es que además de saber realizar bien sus estudios, necesitan conocer en profundidad los programas que va a utilizar, y en muchos casos, necesitan ser ellos mismos los que se programen sus propios códigos.

Un reto que debería asumir la comunidad científica y con ello los fabricantes de ordenadores, sería tratar de separar cada vez más a los científicos de los programas. Con ello los científicos se centrarían únicamente en realizar sus simulaciones y no tratar de optimizar códigos y mejorar las prestaciones de los mismos. Este trabajo debería ser realizado por expertos en los centros de cálculo de cada organización.

Por ello la importancia de este trabajo fin de carrera es doble:

- Por un lado, mostrar a los científicos cómo siguiendo esta metodología, se pueden emplear técnicas simples para acometer la implementación de códigos eficientes desde el principio.

- Por otro lado, mostrar técnicas complejas a los expertos en programación y optimización que debieran existir en cada uno de los centros de cálculo y que sirvieran de soporte a los usuarios científicos.

A día de hoy, los centros de cálculo suelen renovar cada 4 o 5 cinco años sus sistemas de cálculo por completo. Durante el ciclo de vida de estos sistemas, la mayoría suelen ser utilizados al 100%. Sin embargo, muchas veces los códigos que se están ejecutando sobre ellos no son del todo eficientes y trabajos que pudieran ser realizados en horas al final tardan días, impidiendo a la vez que otros usuarios pudieran ejecutar más trabajos.

Por ello, la importancia de este trabajo fin de carrera es enseñar como se puede optimizar la eficiencia en los centros de cálculo empleando técnicas de optimización de códigos.

2.4 ÁMBITO DE APLICACIÓN

En la actualidad soluciones de computación son aplicadas a muchas disciplinas, tanto en el sector privado como en el público:

- Meteorología: Predicción del tiempo que va a hacer en las siguientes horas, días o semanas.
- Climatología: Predicción de un modelo climático a años, décadas, etc.
- Oceanografía: Predicción de un modelo oceanográfico a años, décadas, etc.
- Genómica: Prevención y solución de enfermedades congénitas.
- Biotecnología: Creación de nuevos fármacos.
- Física de los materiales: Dinámica de estructuras para nuevos materiales.
- Química: Dinámica de estructuras de proteínas, etc.
- Compuestos Químicos: Creación de nuevos compuestos químicos.
- Matemáticas: Resolución de modelos matemáticos complejos.
- Finanzas: Modelos matemáticos en análisis de mercados, futuribles, análisis de riesgos, valores bursátiles, etc.
- Oil & Gas: Exploración petrolífera para nuevos yacimientos de petróleo, gas, etc.
- Criptografía y Encriptación: Cifrado y descifrado de mensajes.
- Dinámica de fluidos: Desde el diseño de un pañal hasta el modelado de la superficie de un recipiente.
- Automoción: Aerodinámica para evaluar el impacto del viento sobre la estructura de los coches.
- Automoción: Simulación de choques en coches, aviones, trenes, etc.
- Renderizado: Animación de los efectos de luz y colores sobre objetos animados.
- Industria Naval: Cálculo de estructuras.
- Astrofísica: Modelado del universo, recreando la formación de estrellas y planetas, agujeros negros, etc.
- Termodinámica: Nuevos materiales resistentes a temperaturas extremas.
- Aeroespacial: Diseño de aviones, naves espaciales, trayectorias, etc.
- Neurología: Análisis del cerebro para mapear toda su estructura y funciones.

- Geociencia: Simulando el manto y la tectónica de placas de la tierra para mejorar y comprender el conocimiento sobre terremotos y tratar de anticiparse a ellos.

- Energías renovables.

Como puede apreciarse, los campos de aplicación de esta metodología son amplios.

3 HISTORIA DE LAS ARQUITECTURAS DE COMPUTADORES

Son muchas las referencias que nos ha dejado la historia sobre diferentes inventos que se han realizado en este campo.

Desde tiempos inmemoriales, el ser humano ha querido automatizar los procesos de cálculo que debía realizar manualmente o mentalmente.

La mayoría de las bibliografías consultadas como por ejemplo [Nash, 1990], [Campbell-Kelly, 1996] y [Patterson and Hennessy, 2005], separan la historia de los computadores en cinco generaciones comenzando la primera generación con la aplicación de las teorías de Von Newman a sistemas.

Previos a la primera generación, ha habido multitud de inventos que permitieron a las personas avanzar en la automatización de los cálculos.

Todos los inventos tienen una base común: optimizar los tiempos de cálculo en base a mejorar o bien el diseño de las operaciones que se realizaban, o bien los componentes físicos con los cuales se realizaban estos cálculos.

3.1 SISTEMAS ANTERIORES A LA PRIMERA GENERACIÓN

Desde la antigua Mesopotamia (3000 A.C.) se empezaron a construir sistemas de cálculo primitivos.

Sobre el Siglo VI A.C., aparece el primer dispositivo mecánico utilizado para realizar cálculos, el conocidísimo ábaco.

No fue hasta el Siglo XVII donde se tienen las siguientes referencias de dispositivos de automatización:

- Las tablas neperianas de John Neper (1614).
- La regla de cálculo de Edmund Gunter (1620) basados en los logaritmos de Neper.
- La sumadora de ruedas dentadas de Blaise Pascal (1642).

Figura 1: Rueda Dentada de Pascal. Museo de Ranquet en Clermont-Ferrand (Francia).

- La rueda de Gottfried W. von Leibnitz (1694) que permitía sumar, restar, multiplicar, dividir y calcular raíces cuadradas.

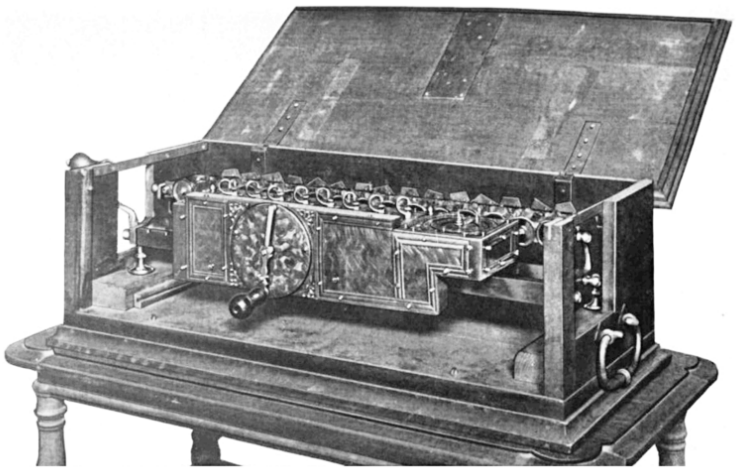

Figura 2: Rueda de Leibnitz. Librería Nacional de Baja Sajonia, en Hannover, (Alemania).

- El telar mecánico de Joseph-Marie Jacquard (1753-1834).

Figura 3: Telar Jacquard en el Museo de la ciencia y la industria, en Manchester (Inglaterra).

- Las máquinas de Charles Babbage (1823-1833).

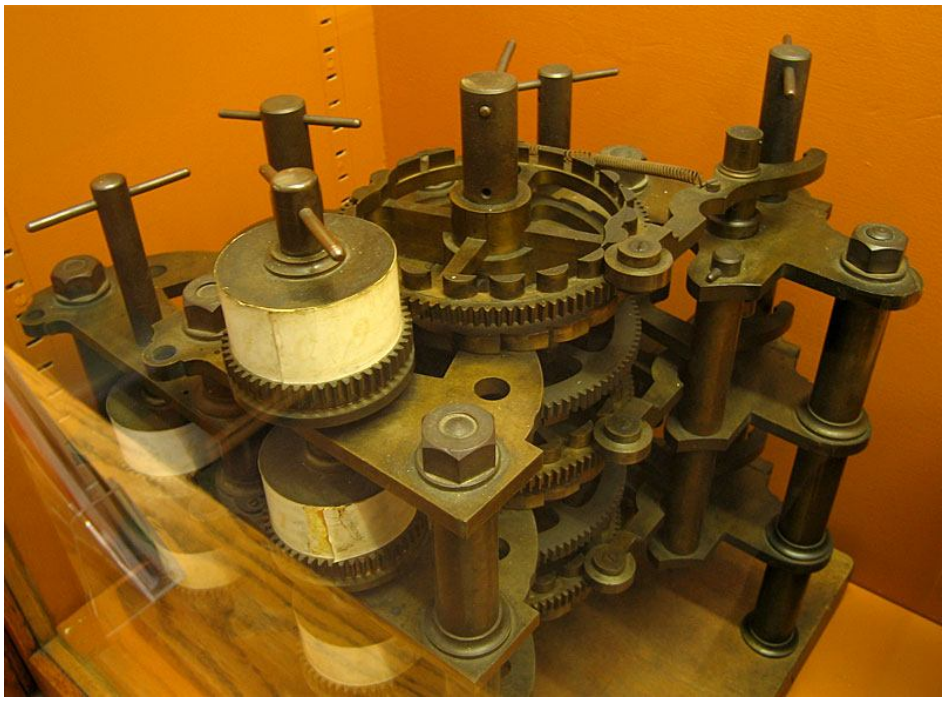

Figura 4: Máquina de Babbage, Museo Whipple de la Universidad de Cambridge (Inglaterra).

- La máquina de sumar y hacer listas de William Burroughs (1882).

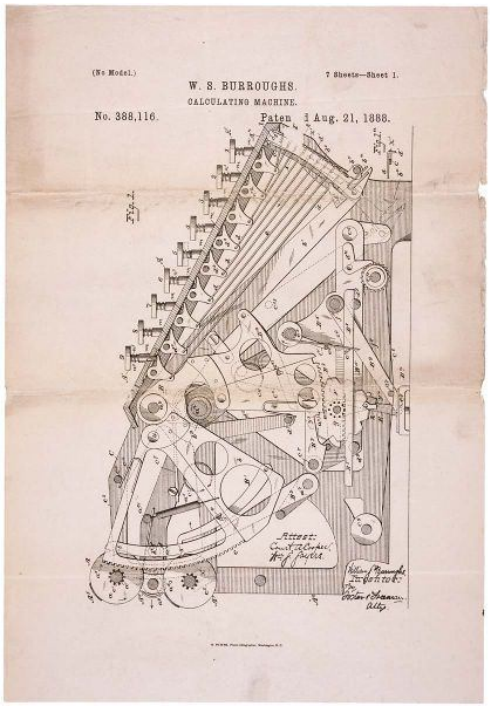

Figura 5: Patente impresa de la Máquina de William Burroughs (EEUU).

- La máquina del censo del fundador de IBM Herman Hollerith (1890).

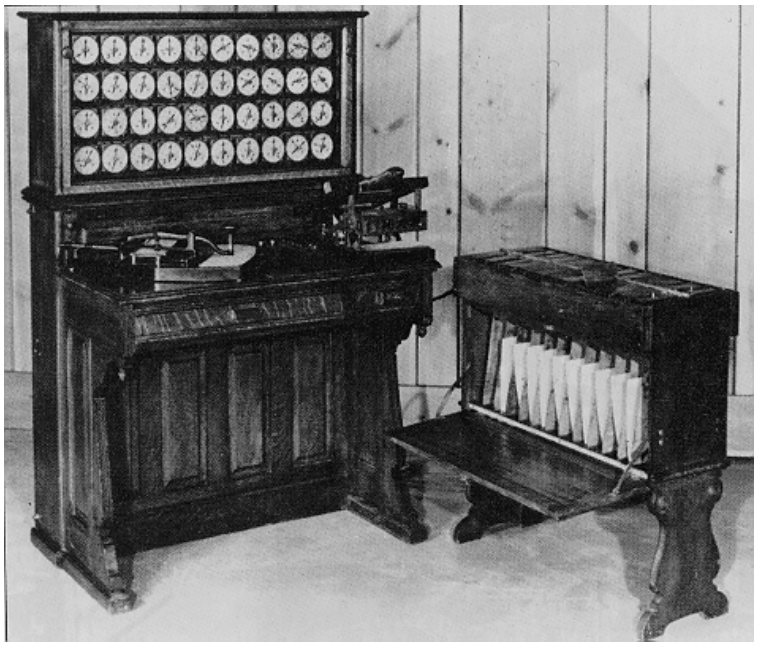

Figura 6: Máquina del Censo de Herman Hollerith.

No obstante, en las primeras décadas del siglo XX fue cuando se obtuvieron diferentes avances teóricos y tecnológicos que posibilitarían la aparición de los primeros ordenadores. De esta forma fueron apareciendo sucesivamente:

- Z1 y sus revisiones posteriores Z2, Z3 y Z4 de Konrad Zuse durante la segunda guerra mundial, que fue la primera máquina calculadora que se basaba en reveladores.

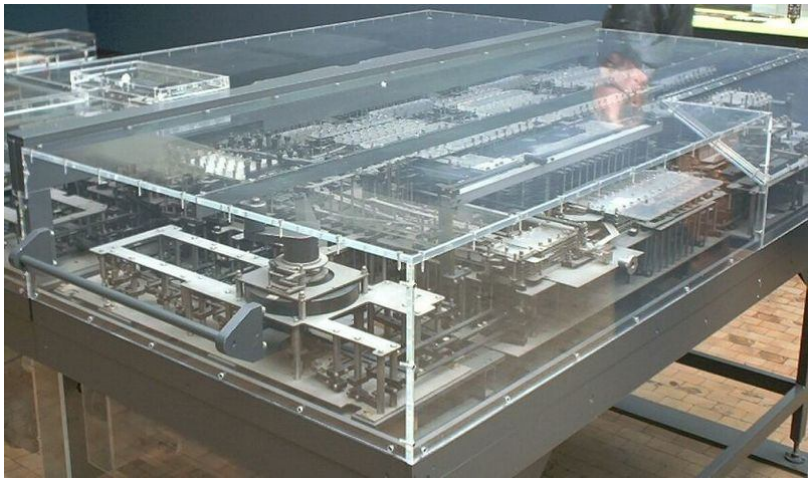

Figura 7: Sistema Z1 de Zuse en el Museo Técnico de Berlín.

- Harvard Mark I de Howard Aiken (1939-1944), que se puede considerar como la primera máquina de propósito general.

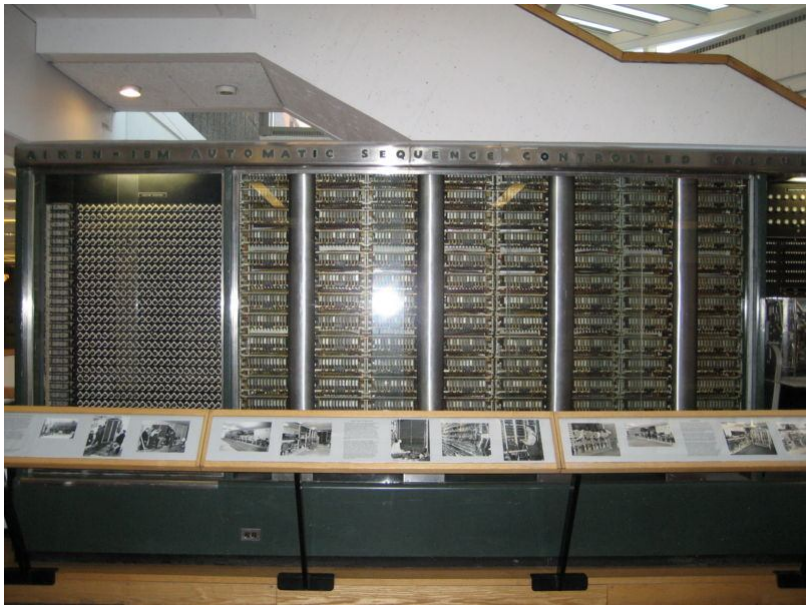

Figura 8: Sistema Harvard Mark I en el Edificio de Ciencias Cabot, (USA).

- Colossus del gobierno británico bajo la supervisión de Alan Turing (1943), utilizado para descifrar los mensajes alemanes durante la segunda guerra mundial.

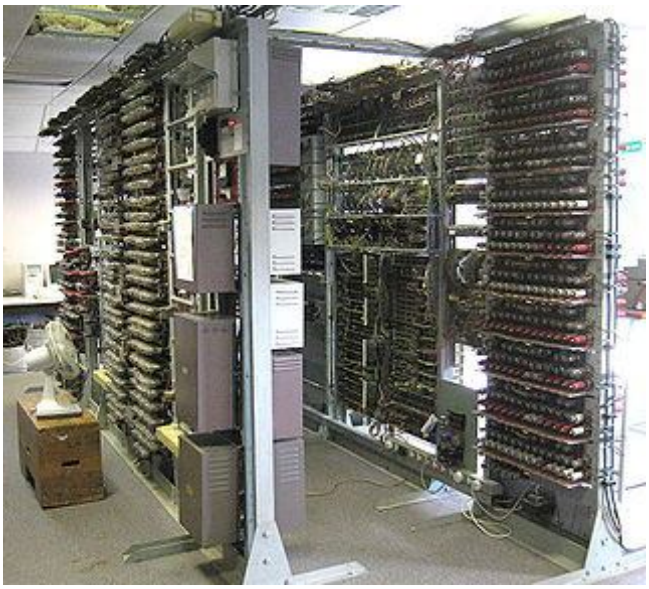

Figura 9: Reproducción del Sistema Colossus en Buckinghamshire (Inglaterra).

- ENIAC I de John Eckert y John Mauchly de 1948, primer ordenador de válvulas que a pesar de su gran tamaño (30 toneladas) permitía realizar 5.000 operaciones aritméticas en 1 segundo y se le podía reprogramar las tareas a realizar cambiando cables y conexiones. Sin embargo tenía cerca de 18.000 tubos de vacío y consumía más de 200 KW.

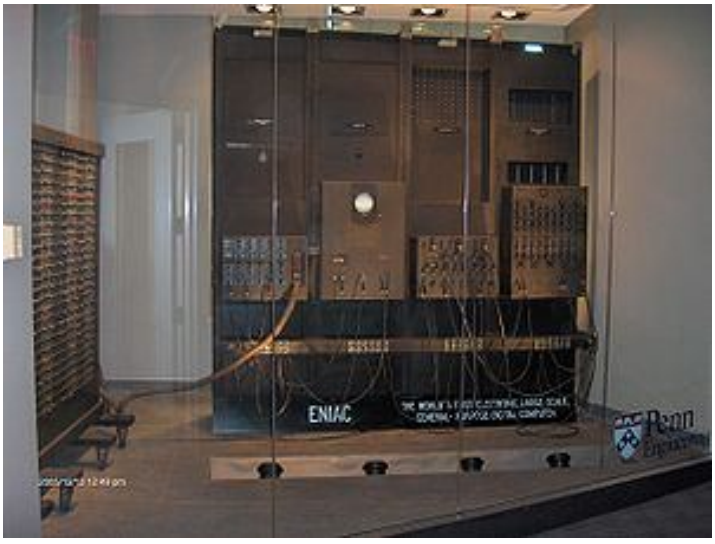

Figura 10: Sistema ENIAC en la Universidad de Pennsylvania (EEUU).

- EDSAC de Wikes de 1949 que permitía almacenar los programas en memoria.

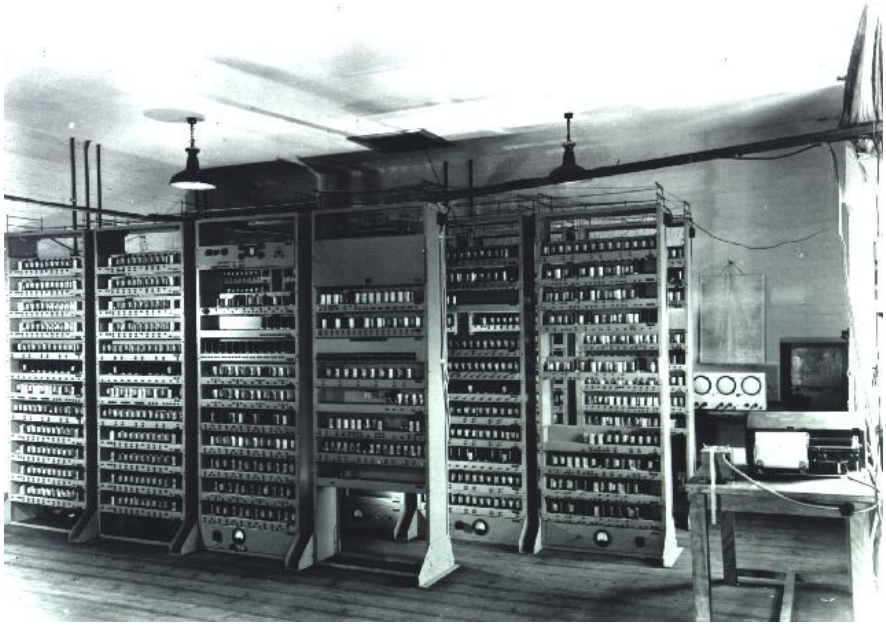

Figura 11: Sistema EDSAC, Universidad de Cambridge (Inglaterra).

No obstante, fue John Von Newman el que sentó las bases de los ordenadores modernos al proponer una arquitectura de computadores que ha perdurado hasta hoy en día.

John Von Newman determinó que se debía reemplazar la aritmética decimal utilizada en la computadora ENIAC por una aritmética binaria y que además, se debían utilizar "memorias" donde deberían residir las instrucciones que se debían ejecutar conjuntamente con los datos sobre los cuales se operara.

La máquina de Von Newman tenía cuatro partes principales: la memoria, la unidad de control, la unidad aritmética-lógica y la unidad de entrada y salida.

De esta forma, se sentaron también las bases para la creación de lenguajes que permitieran programar los computadores.

Von Newman realizó un diseño básico de su mismo nombre (la máquina de Von Newman) que se usó en la construcción de la computadora EDVAC (*Electronic Discrete Variable Automatic Computer*) considerada por entonces como la primera computadora que almacenaba el programa. Tenía más de cuatro mil válvulas y usaba un tipo de memoria basado en tubos llenos de mercurio.

Basándose en estos principios, surgió todo un desarrollo de computadores que tradicionalmente se han dividido en cinco generaciones.

3.2 PRIMERA GENERACIÓN

Esta generación abarcó en tiempo más o menos la década de los años cincuenta.

Los sistemas de esta generación tenían como principales características:

- Estaban construidos por medio de tubos de vacío.
- Eran programados en lenguaje máquina.

En 1951 J. Presper Eckert y John William Mauchly construyen la UNIVAC (*UNIVersAl Computer*) considerada tradicionalmente como la primera computadora comercial. Este sistema disponía de más de mil palabras de memoria central e incluso podía leer cintas magnéticas.

El primer UNIVAC se utilizó para realizar el censo electoral en Estados Unidos, aunque se vendieron más, tanto al sector público como al privado.

Figura 12: Sistema UNIVAC, Museo Técnico de Viena (Austria).

Herman Hollerith el que fuera fundador de IBM (*International Bussines Machines*), incorporó una característica fundamental a estos sistemas: el uso de tarjetas perforadas a través de unidades de entrada. Con esta idea, se comercializó el modelo IBM 701 y 702 que compitieron con el modelo 1103 de Remington Rand.

Figura 13: Sistema IBM 701, IBM Headquarter en Nueva York (EEUU).

No obstante, la computadora más popular de la primera generación fue el modelo IBM 650, del cual se comercializaron cientos de unidades. El modelo IBM 650 introdujo un elemento revolucionario como era el uso de memoria secundaria mediante un tambor magnético, que es el antecesor de los discos actuales.

Figura 14: Tambor magnético del Sistema IBM 650.

Citar también otros modelos de computadoras que se sitúan entre la primera y segunda generación como son la UNIVAC 80 y 90, las IBM 704 y 709, Burroughs 220 y UNIVAC 1105.

En cuanto a la programación, en 1951 la Universidad de California en Livermore desarrolló un lenguaje de compilación y de ejecución denominado KOMPILER para el sistema IBM 701. Esto fueron los orígenes del compilador de Fortran (*The IBM Mathematical **For**mula **Tran**slating System*) que fue producido por IBM para el sistema IBM 704.

3.3 SEGUNDA GENERACIÓN

En la década de 1960, las computadoras redujeron su tamaño y consumo y crecieron su capacidad de procesamiento.

Las características de la segunda generación son las siguientes:

- Están construidas en base a circuitos de transistores.
- Se programan en lenguajes de alto nivel.

Aparecen nuevas compañías que se dedican a la construcción de computadoras. Algunas como la serie 5000 de Burroughs eran bastante avanzadas para su época.

Las computadoras de la década de los 60 tenían que ser programadas por un equipo de expertos que debían resolver los problemas y los cálculos solicitados por la administración. Los sistemas estaban limitados a aquellas personas que sabían programar y se necesitaba dedicar muchas horas escribiendo instrucciones, ejecutando el programa resultante y depurando y corrigiendo los errores que iban apareciendo.

En esta época se comienzan a desarrollar teorías de algorítmica. Edsger Dijkstra elabora un algoritmo eficiente para hallar las rutas más cortas en grafos como una demostración de las cualidades de la computadora ARMAC.

En 1960 C. Antony R. Hoare desarrolla un algoritmo de ordenamiento llamado quicksort.

Desde entonces, la algorítmica ha ido mejorando y reduciendo las horas de cálculo destinadas a resolver problemas complejos.

En esta época se debía guardar de alguna forma el programa resultante para no perderlo puesto que se había tardado muchas horas en escribirlo. Para tal propósito, se usaban grabadoras cuyo procedimiento podía tomar entre 10 y 45 minutos, dependiendo del programa.

Algunas computadoras de esta generación fueron la Control Data Corporation (creada por Seymour Cray) modelo 1604, Philco 212, la UNIVAC M460, IBM 7090, NCR 315, RCA 501 y 601.

Figura 15: Modelo IBM 709.

La Radio Corporation of America (RCA) comercializó el modelo 501 y 601, que podía manejar el lenguaje COBOL.

En cuanto a programación, en 1960 se crea el lenguaje COBOL el cual ha sido el lenguaje de programación más utilizado hasta la fecha.

En el mismo año, se crea Algol, que fue el primer lenguaje estructurado y APL, orientado a trabajo con matrices.

3.4 TERCERA GENERACIÓN

Con la aparición del modelo IBM 360 en Abril de 1964 que utilizaba circuitos integrados, surge la tercera generación de las computadoras.

Las características de esta generación fueron las siguientes:

- Su fabricación electrónica esta basada en circuitos integrados.
- Se manejan a través de lenguajes de control denominados sistemas operativos.
- Aparición de software comercial.

La serie IBM 360 utilizaba unidades de cinta, técnicas especiales del procesador y discos magnéticos. Para su control, utilizaba un técnica revolucionaria mediante el uso de un sistema operativo, que se llamó OS.

Figura 16: IBM 360 modelo 40.

En 1964 apareció el modelo 6600 de CDC que durante algunos años se consideró como el computador más rápido.

Robert Noyce y Gordon Moore fundan en 1968 la Corporación Intel cuyos procesadores serán los más vendidos décadas después. Es famosa la ley de Moore en la cual determinó que cada 16 meses se iba a duplicar el número de transistores en los computadores cumpliéndose dicha ley prácticamente hasta la fecha actual.

Ken Thompson y Dennis Ritchie desarrollan en los laboratorios Bell un lenguaje de programación denominado B, que fue la base para el desarrollo de uno de los lenguajes de programación más difundidos hasta la fecha: el lenguaje C.

Junto con Douglas Mcllory y basándose en B, desarrollan un sistema operativo denominado Unix que será el sistema operativo por excelencia durante las siguientes décadas.

Ya en la década de los 70, IBM evoluciona su serie con la aparición de la 370. Otros sistemas de esta época son UNIVAC 1108 y 1110, CDC 7600 y la serie 6500 y 6700 de Burroughs.

Figura 17: IBM 370 modelo 168.

Fue a mediados de la década de 1970 cuando los computadores empezaron a utilizarse para mayor consumo, puesto que tenía un tamaño mediano y no eran tan costosos.

Estos sistemas se denominaron minicomputadoras. Ejemplos de estas fueron las siguientes: la serie 3000 y 9000 de Hewlett – Packard, PDP 8 y 11 y VAX de DEC, modelos NOVA y ECLIPSE de Data General y la Wang de Honey - Well - Bull.

3.5 CUARTA GENERACIÓN

A mediados de los años 70, aparecen los microprocesadores y con ella la cuarta generación de computadores.

Las características principales de esta generación fueron las siguientes:

- Circuitos Integrados.
- Aparición de compañías exclusivas de software.
- Estandarización de la informática como bien de consumo empresarial.

Este avance revolucionario en el campo de la microelectrónica conlleva la aparición de los primeros circuitos integrados.

Los computadores basados en circuitos integrados permiten llevar la computación al mercado de consumo. Los sistemas creados con circuitos integrados eran

extremadamente pequeños y baratos, por lo que se podía destinar no sólo a cálculo científico sino que se podían utilizar en el mercado industrial de consumo.

En 1975 la revista Popular Electronics vende el Altair 8800 basado en el procesador Intel 8080A y que tradicionalmente ha sido considerado el primer computador personal de consumo. Utilizaba el lenguaje de programación Altair BASIC, de Bill Gates y Paul Allen, fundadores de Microsoft.

Figura 18: Altair 8800.

En 1977 los fundadores de Apple Steve Wozniak y Steve Jobs inventan el primer ordenador de consumo: el Apple II. Funcionaba bajo el procesador 6502 a la velocidad de 1 Mhz.

Figura 19: Apple II.

Con la aparición de los ordenadores personales, se necesitaba facilitar su uso al público en general.

Por ello, aparecieron aplicaciones de propósito general como el procesador de textos Word Star y la hoja de cálculo Visicalc.

El software comienza a tener un peso igual de importante que el hardware.

IBM viendo el potencial mercado de los ordenadores de consumo, lanzó en Agosto de 1981 el modelo IBM 5150 que supondría la confirmación de los computadores para uso de consumo en masa. Se decidió construir un sistema basado en arquitectura abierta con componentes estándar en el mercado para que otros fabricantes pudieran emular el sistema y sacar al mercado sistemas compatibles con él.

Figura 20: IBM PC 5150.

El número de ordenadores personales vendidos cada año fue subiendo exponencialmente, desde 80.000 en 1981 hasta más de 60 millones hasta el 1987.

Desde entonces, se dividieron los computadores en 2 grandes ramas: ordenadores personales o PC para consumo de público general y pequeña y mediana empresa y grandes computadores para cálculo.

CDC, CRAY, Silicon Graphics, NEC, Hitachi o IBM se centran en sistemas grandes de computación capaces de atender a varios cientos de millones de operaciones por segundo y a cientos de usuarios simultáneamente.

CDC STAR-100 y IILLIAC IV se convierten en los primeros sistemas vectoriales que aparecen en el mercado. Ya en los años 60, bajo el proyecto Solomon, se habían sentado las bases para que desde un solo procesador se pudieran manejar más de una unidad aritmética (ALU), lo que permitía aplicar un algoritmo simple a un gran conjunto de datos. Esto es debido a que los códigos científicos trabajan sobre vectores y matrices. El CDC STAR-100 se convirtió en el primer sistema que llegaba a los 100 Mflops.

Seymur Cray deja CDC y funda su propia compañía Cray Research, lanzando el sistema Cray-1 de 80 Mhz en 1976. A este, le sucedió el primer sistema multiproceso, el Cray X-MP de 800 Mflops en 1982.

Figura 21: Cray-1.

3.6 QUINTA GENERACIÓN

Esta generación ha visto la llegada de los ordenadores a la sociedad de consumo. Actualmente no se concibe la vida ni el trabajo sin esta herramienta. La evolución de la microelectrónica llevó también la evolución del software de consumo y surgen muchas compañías exclusivas de software como Microsoft, Novell, Lotus, Borland, etc.

Las características principales de esta generación fueron las siguientes:

- Llegada masiva de los ordenadores a todos los hogares.
- Sistemas masivamente paralelos.
- Internet como herramienta de comunicación global.

En el ámbito internacional, Japón y EUU compiten por el dominio del mercado, desarrollando programas de investigación con el objetivo de crear sistemas productivos que se acercaran más al lenguaje natural del ser humano.

En la década de los 80, Cray saca al mercado sistemas vectoriales y paralelos como el Cray-2, Y-MP (considerado el primer sistema que pasó del Gigaflop real), C90 y T90.

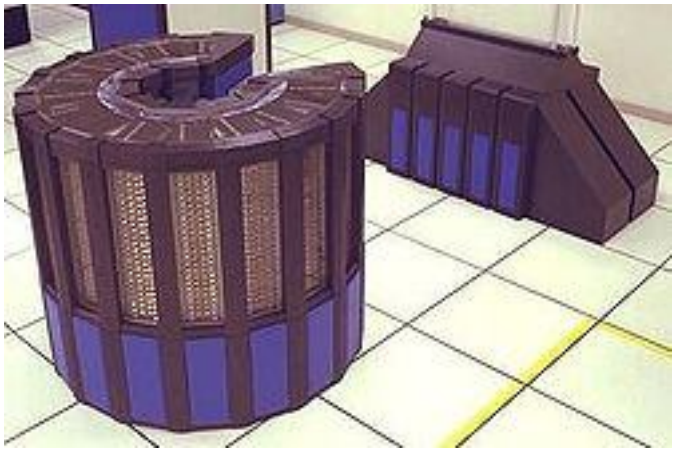

Figura 22: Cray-2.

Silicon Graphics nacida en 1982, emerge como líder en computación sacando al mercado los modelos Power y Challenge.

NEC en Japón se convierte en otro líder indiscutible con la salida de los sistemas SX: SX1 en 1983, SX2 y SX3 en 1989 hasta la familia SX9 en la actualidad.

Ya en la década de los 90, IBM saca al mercado una nueva generación de procesadores que han llegado a convertirse en los más rápidos: la familia Power, desde la 1 hasta la actual Power 6.

En 1991, Linus Torvalds escribe su propio sistema operativo Linux, el cual se convertiría en la base de los sistemas de computación del siglo XXI, y lo que es más importante, la idea de Open Source que permitiría el desarrollo de código abierto que será la base de colaboración y evolución de código científico y de programas en el mundo.

En esta década, Compaq dominó el mercado de la computación de consumo medio, con su procesador Alpha.

En Japón, NEC evoluciona su línea SX con la aparición de los sistemas SX-4 y SX-5. Compitiendo con NEC, Hitachi saca al mercado su modelo vectorial Hitachi SR2201 al igual que Fujitsu con su línea VP2600 y S400.

Es esta la década donde la presencia de grandes sistemas vectoriales era la nota predominante en la mayoría de los grandes centros de investigación de todo el mundo.

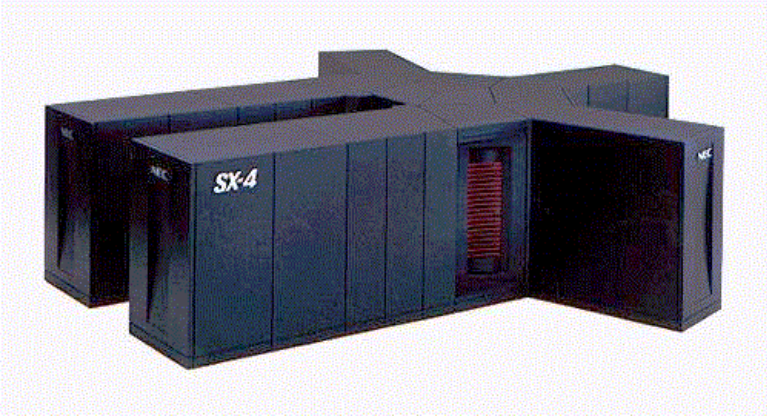

Figura 23: NEC SX-4.

En 1993 aparece una lista con el objetivo de clasificar los 500 sistemas más potentes del mundo, el top500 (www.top500.org) basado en el benchmark de Linpack de John Longarra. El primer sistema sería ocupado por el modelo de Thinking Machine's Connection CM-5 de 1024 procesadores con una potencia pico de más de 130 Gflops adquirida posteriormente por SUN Microsystems.

En 1994, Intel instala el modelo Paragon en los laboratorios Sandia, el primer sistema cluster de memoria distribuida con 3680 procesadores.

IBM saca al mercado sus modelos SP2 basados en el procesador Power.

Figura 24: IBM SP-2.

En 1996, Silicon Graphics compra Cray sacando al mercado sus primeros sistemas SN1 basados en NumaLink, el Origin 2000 los cuales podían llegar a escalar hasta 128 procesadores con 256 GB de memoria compartida. Silicon Graphics dejó su línea 10.000 a SUN que la convertiría en un gran éxito de ventas.

Figura 25: SGI Origin 2000.

En 1997 se instala en los laboratorios Sandia de Estados Unidos el sistema Intel ASCI Red, otro sistema Intel Paragon basado en Intel Pentium Pro a 200 Mhz, convirtiéndose en el primer sistema en alcanzar el Tflop y permaneciendo como el sistema de mayor potencia del mundo hasta Noviembre del 2000. Los cluster de Intel se convierten en sistemas líderes en computación.

Figura 26: Intel 860 Paragon XP.

Cray a su vez, saca al mercado la línea de sistemas T3E.

Figura 27: Cray T3E.

En Noviembre del 2000, IBM consigue convertirse por primera vez desde hacía décadas, en el proveedor con el sistema más potente del mercado por su instalación en el Laboratorio Nacional de Lawrence Livermore de un cluster de SP Power 3.

Figura 28: IBM ASCI White.

A su vez, SGI saca al mercado los sistemas Origin 3000 llegando a alcanzar 512 procesadores con una sola imagen de sistema operativo (SSI).

Los sistemas UNIX copan el mercado de la computación mientras que Windows domina en el mercado de consumo.

Los procesadores RISC y vectoriales copan prácticamente todo el mercado mientras que Intel y Linux van cogiendo fuerza.

En cuanto a arquitectura, los cluster empiezan a colocarse por encima de los sistemas únicos de memoria compartida.

En el año 2002, HP compra Compaq convirtiéndose en uno de los líderes de HPC junto con IBM, que pierde su hegemonía a la vez que Estados Unidos con el sistema Earth-Simulator de NEC en Japón, que con un pico de 40 Tflops rompe las barreras de prestaciones hasta entonces fijadas. Este sistema sería el último sistema vectorial dominante puesto que desde entonces, los sistemas vectoriales acabarían prácticamente desapareciendo del mercado siendo reemplazados por sistemas en cluster.

Figura 29: Earth Simulator en Japón.

Por primera vez, Europa consigue situar un sistema entre los 5 más potentes del mundo con la instalación de BULL en el CEA.

A partir del 2003, Intel y Power dominarán el mercado de la computación hasta ahora, mientras que Linux se impone como sistema operativo por excelencia.

3.7 ¿SEXTA GENERACIÓN?

Desde la aparición de tarjetas gráficas para el mercado de consumo a finales de los años 90, se empezó a analizar la posibilidad de utilizarlas para el mercado de computación. Esta ciencia se conoce con el nombre de GPGPU (*General-Purpose Computing on Graphics Processing Units* o Unidades de Procesamiento Gráfico para Computación de Propósito General). La potencia y el número de transistores que se utilizan en las tarjetas gráficas son muy superiores al de los procesadores tradicionales. Una tarjeta gráfica es capaz de procesar millones de vértices y miles de fragmentos por segundo. Además su arquitectura es altamente paralela al aplicarse el modelo SIMD (*Single*

Instruction Multiple Data) de tal forma que todas sus unidades ejecutan una misma instrucción simultáneamente.

En el año 2001 Sony Computer Entertainment, IBM y Toshiba crearon una alianza conocida con el nombre de "STI" para el desarrollo de una nueva arquitectura de microprocesador denominada Cell, que empleaba una combinación de tarjetas coprocesadoras gráficas con arquitectura del procesador PowerPC.

El primer desarrollo comercial de Cell fue la conocida videoconsola PlayStation 3 de Sony. En el campo de la computación, IBM ha centrado su solución de HPC en torno a este procesador.

Su gran competidor en este terreno es NVIDIA. Desde el año 2004 se empezó a profundizar en esta materia con la aparición de tarjetas gráficas convencionales que permitían una programación más sencilla.

En ese momento, NVIDIA y ATI son los 2 fabricantes dominadores en el mercado. Sin embargo, NVIDIA presenta una evolución en el entorno de programación denominado CUDA que la ha aupado al dominio actual en este sector con respecto a su competidor ATI.

Con la aparición de la tarjeta NVIDIA Geforce 6 en el 2004 que incluía 16 unidades de cálculo se empezó a generalizar su utilización en el mundo de la computación.

En 2005, la tarjeta Geforce 7 incluía 24 unidades de cálculo.

Fue ya en 2006 con la aparición de la tarjeta Geforce 8 de 128 unidades de coma flotante cuando la programación GPGPU tomó fuerza. Se desarrolla una suite completa de desarrollo denominada CUDA que incluye compilador, herramientas de prestaciones, debugging, librerías científicas (BLAST y FFT), etc.

Sin embargo, tanto Cell como la Geforce 8 tenían un hándicap para el cálculo científico: sus unidades de coma flotante son de precisión simple, mientras que los cálculos científicos se desarrollan en su inmensa mayoría en doble precisión.

Con la aparición en 2008 de la familia GT200 con doble precisión (240 unidades de coma flotante de precisión simple y 30 de doble precisión), el mundo científico ha encontrado una plataforma de aceleración de códigos hasta ahora no vista. A partir de 2010 las tarjetas GPU dispondrán en todas sus unidades de precisión doble.

Según NVIDIA, la siguiente generación de tarjetas tendrá un rendimiento pico en coma flotante de doble precisión de 800 Gflops, 8 veces más que un servidor actual con 2 procesadores de cuádruple núcleo a 3 Ghz (96 Gflops pico por servidor).

Con la aparición de las GPU en el mercado de computación, las prestaciones en los códigos aumentarán a una velocidad vertiginosa.

IBM RoadRunner basado en Cell fue el primer sistema que llegó al Petaflop.

Figura 30: IBM RoadRunner.

Intel, no queriendo perder el dominio en el mundo de los microprocesadores, sacará en el año 2010 su propia versión de tarjeta GPU, conocida como Larrabee y que presentará según Intel, grandes mejoras con respecto a NVIDIA y Cell a nivel de programación.

NVIDIA, ATI, IBM e Intel dominarán el mercado de computación los próximos años, en lo se puede denominar una Sexta Generación de Computadores.

En el siguiente cuadro se hace un resumen de los computadores que se han visto anteriormente y su rendimiento en Flops.

Año	Nombre	Velocidad Pico (Rmax)	Lugar
1942	Atanasoff–Berry Computer (ABC)	30 OPS	Iowa State University, Ames, Iowa, USA
	TRE Heath Robinson	200 OPS	Bletchley Park, Bletchley, UK
1944	Flowers Colossus	5 KOPS	Post Office Research Station, Dollis Hill, UK
1946	UPenn ENIAC	100 KOPS	Departamento de Guerra

			Aberdeen Proving Ground, Maryland, USA
1954	IBM NORC	67 KOPS	Departamento de Defensa U.S. Naval Proving Ground, Dahlgren, Virginia, USA
1956	MIT TX-0	83 KOPS	Massachusetts Inst. of Technology, Lexington, Massachusetts, USA
1958	IBM AN/FSQ-7	400 KOPS	25 departamentos de U.S. Air Force (52 computers)
1960	UNIVAC LARC	250 KFLOPS	Atomic Energy Commission (AEC) Lawrence Livermore National Laboratory, California, USA
1961	IBM 7030	1.2 MFLOPS	AEC-Los Alamos National Laboratory, New Mexico, USA
1964	CDC 6600	3 MFLOPS	AEC-Lawrence Livermore National Laboratory, California, USA
1969	CDC 7600	36 MFLOPS	
1974	CDC STAR-100	100 MFLOPS	
1975	Burroughs ILLIAC IV	150 MFLOPS	NASA Ames Research Center, California, USA

1976	Cray-1	250 MFLOPS	Energy Research and Development Administration (ERDA) Los Alamos National Laboratory, New Mexico, USA
1981	CDC Cyber 205	400 MFLOPS	Múltiples instituciones en todo el mundo
1983	Cray X-MP/4	941 MFLOPS	U.S. Department of Energy (DoE) Los Alamos National Laboratory; Lawrence Livermore National Laboratory
1984	M-13	2.4 GFLOPS	Scientific Research Institute of Computer Complexes, Moscow, USSR
1985	Cray-2/8	3.9 GFLOPS	DoE-Lawrence Livermore National Laboratory, California, USA
1989	ETA10-G/8	10.3 GFLOPS	Florida State University, Florida, USA
1990	NEC SX-3/44R	23.2 GFLOPS	NEC Fuchu Plant, Fuchu, Japan
1993	Thinking Machines CM-5/1024	65.5 GFLOPS	DoE-Los Alamos National Laboratory; National Security Agency

	Fujitsu Numerical Wind Tunnel	124.50 GFLOPS	National Aerospace Laboratory, Tokyo, Japan
	Intel Paragon XP/S 140	143.40 GFLOPS	DoE-Sandia National Laboratories, New Mexico, USA
1994	Fujitsu Numerical Wind Tunnel	170.40 GFLOPS	National Aerospace Laboratory, Tokyo, Japan
1996	Hitachi SR2201/1024	220.4 GFLOPS	University of Tokyo, Japan
	Hitachi/Tsukuba CP-PACS/2048	368.2 GFLOPS	Center for Computational Physics, University of Tsukuba, Tsukuba, Japan
1997	Intel ASCI Red/9152	1.338 TFLOPS	DoE-Sandia National Laboratories, New Mexico, USA
1999	Intel ASCI Red/9632	2.3796 TFLOPS	
2000	IBM ASCI White	7.226 TFLOPS	DoE-Lawrence Livermore National Laboratory, California, USA
2002	NEC Earth Simulator	35.86 TFLOPS	Earth Simulator Center, Yokohama, Japan
2004	IBM Blue Gene/L	70.72 TFLOPS	DoE/IBM Rochester, Minnesota, USA
2005		136.8 TFLOPS	DoE/U.S. National Nuclear

		280.6 TFLOPS	Security Administration, Lawrence Livermore National Laboratory, California, USA
2007		478.2 TFLOPS	
2008	IBM Roadrunner	1.026 PFLOPS	DoE/U.S. Los Alamos, Nuevo Mexico. USA

Figura 31: Rendimiento de los sistemas a lo largo de la historia.

Se estima que a finales de la segunda década se consiga el primer sistema ExaFlop mientras que un sistema ZettaFlop se alcanzará a mediados de la tercera década, sobre el año 2025.

4 ARQUITECTURA DE LOS COMPUTADORES Y CONCEPTOS BÁSICOS

4.1 INTRODUCCIÓN

El conocimiento de la arquitectura que se está utilizando en el desarrollo y ejecución de los códigos, es fundamental para optimizar los tiempos de ejecución de los mismos.

Como se ha descrito en el capítulo anterior, a lo largo del siglo pasado y del presente, han ido surgiendo diferentes tipos de arquitecturas, procesadores, memorias, etc.

Diseñar un código óptimo, pasa siempre por una primera fase de pensar en que tipo de computador se va a ejecutar.

Adicionalmente, se debe pensar en el futuro de la plataforma que se va utilizar. Si se va a implementar un código sobre una plataforma obsoleta o con pocas vías de evolución, se podrá desarrollar un código que sea muy bueno a día de hoy pero que no podrá ser utilizado en los próximos años debido a que simplemente esa plataforma, procesador, sistema operativo, etc. dejará de estar soportado o no evolucionará.

Las empresas de hardware siguen investigando en computadores que les permita diferenciarse del resto de competidores del mercado. Para cada uno de los diferentes fabricantes el componente hardware constituye una parte fundamental en el campo de investigación de los futuros productos que pondrán en el mercado, puesto que les ofrecerá elementos distintivos desde el punto de visto competitivo con el resto de fabricantes.

Adicionalmente será preciso disponer de un sistema operativo eficaz y versátil que permita manejar el sistema de manera cómoda, simple y efectiva, y por otro lado que los compiladores y utilidades de desarrollo permitan acercar lo más posible el rendimiento real a los límites que el hardware impone (prestaciones ideales o “rendimiento pico”).

Los computadores actuales están constituidos de componentes tales como los procesadores, memoria, elementos de interconexión, dispositivos periféricos y elementos aceleradores (como FPGA o GPU). Cómo se distribuyen y conectan estos elementos forma una ciencia conocida como Arquitectura de Computadores.

Es por ello que se debe buscar desarrollar en plataformas estándares que se sepa van a continuar su evolución en el mercado y que van a facilitar la migración futura a una nueva versión de procesador, sistema operativo, etc.

4.2 CONCEPTOS BÁSICOS

Es importante para un programador conocer una serie de conceptos básicos de arquitectura de computadores para saber adaptar su código al hardware sobre el cual se va a ejecutar.

4.2.1 PROCESADOR

El procesador es, en todo caso, el cerebro principal de la ejecución de un código. La mayor parte del tiempo de ejecución depende de sus características. Existe toda una teoría de creación de procesadores y muchas compañías han desaparecido del mercado por quedarse atrás en la evolución de su procesador. En la presente década, han desaparecido procesadores como Alpha o PA-RISC.

En la actualidad, el mundo de la computación está dominado principalmente por:

- IBM con la familia Power (Power 6 actualmente).
- INTEL con la familia Itanium 2 y Xeon.
- SUN con SPARC.
- AMD con Opteron.
- Apple/IBM con PowerPC.

De hecho, muchas compañías de hardware disponen actualmente de productos con diferentes familias y tipos de procesadores de los anteriormente mencionados.

El procesador está compuesto básicamente por:

- La Unidad de Control: decide que es lo que hay que hacer en el siguiente paso: cargar datos de memoria, sumar dos valores, guardar datos en memoria, decidir entre dos posibles caminos cuál es el mejor (salto), etc.
- Las unidades aritméticas/lógicas: realizan los cálculos tales como sumas, restas, multiplicaciones, operaciones lógicas (por ejemplo si dos valores son iguales), etc.
- Los registros: donde residen los operandos desde los cuales se van a realizar las operaciones o donde se van a guardar los resultados de dichas operaciones.

4.2.1.1 CICLO DE RELOJ

Dentro de cada procesador hay un pequeño reloj que da pulso cada cierta frecuencia. A cada uno de estos pulsos se le denomina ciclo de reloj o ciclo simplemente.

Se define ciclo de reloj entonces al tiempo que toma un componente como puede ser un procesador o la memoria en realizar una operación básica.

La medida universal son Hertzios (Hz) que es la unidad de frecuencia en el sistema internacional. Un Hertzio es igual a un ciclo por segundo. El nombre viene en honor a Heinrich Hertz, uno de los primeros investigadores de las transmisiones de ondas radio.

Si un procesador funciona a 3 Ghz, un ciclo de reloj será 1/3.000.000.000 segundos.

Hablando de sistemas métricos, es importante comprender las medidas típicas en computadores. Mientras que las unidades de almacenamiento (disco, memoria, etc.) se miden en exponenciales de 2, la frecuencia se mide en exponenciales de 10.

Así encontraremos diferencias entre por ejemplo un Ghz y un GByte.

Un bit (acrónimo de **B**inary Dig**it**) es la unidad mínima de información en el sistema de numeración binario (0 y 1).

Al conjunto de 8 bits se le denomina byte (B).

K es igual a 2^10 bytes o 10^3. Por ello un KB es 1024 Bytes

M (Mega) es igual a 2^20 bytes o 10^6. Por ello un MB es 1024 KB y un Mhz es 1/1.000.000.

G (Giga) es igual a 2^30 o 10^9. Por ello 16 GB son 16*1024 MB y 3 Ghz es 1/3.000.000.000.

T (Tera) es 2^40 o 10^12.

Y así sucesivamente se tendría P (Peta), E (Exa), Z (Zetta), Y (Yotta) y X (Xera).

En líneas generales, cuanta más frecuencia tenga un procesador, más rápida será la ejecución si comparamos frecuencias en un mismo tipo de procesador.

Y adicionalmente, el tiempo que toma en ejecutarse un código, muchas veces es directamente proporcional a la frecuencia de reloj. Aunque conviene destacar que no siempre es así, como se verá posteriormente, puesto que hay muchos otros parámetros que influyen en el rendimiento.

4.2.1.2 INSTRUCCIÓN

Una instrucción es la acción que debe realizar un procesador. Cada procesador lleva "grabado" todas las acciones que puede realizar en lo que se denomina juego o conjunto de instrucciones. Cada procesador lleva implementado los comandos que puede entender y ejecutar.

A grosso modo, se puede decir que existen 3 tipos de juegos de instrucciones:

- CISC (*Complex Instruction Set Computer*)
- RISC (*Reduced Instruction Set Computer*)
- SISC (*Specific Instruction Set Computer*).

Existe un cuarto tipo de juego de instrucciones creado por Intel para su modelo Itanium 2 denominado EPIC (*Explicitly Parallel Instruction Computing*).

4.2.1.3 MIPS Y FLOP

El término MIPS (Millones de Instrucciones por Segundo) hace referencia al número de instrucciones que puede realizar un ordenador por segundo y mide por ello la velocidad de un computador.

Esta medida fue utilizada en el siglo anterior y en la actualidad ha sido reemplazada por otra medida: FLOP (*Floating point Operations Per Second)*, número de operaciones en coma flotante que puede realizar un computador por segundo.

Esta medida es proporcional a la frecuencia del reloj y al número y tipo de FPU (*Floating Point Arithmetic* o unidades de punto o coma flotante) que tenga el procesador.

Los procesadores actuales son capaces de ejecutar hasta 4 operaciones de coma flotante por ciclo de reloj, que generalmente suele ser 2 operaciones FMA (*Floating Multiply Add* o unidad de multiplicación-suma): a=b*c+d.

Los procesadores estándares (Xeon, Itanium 2 o Power 6) tienen 2 unidades FPU, cada una de las cuales puede hacer una operación FMA por ciclo de reloj.

De este forma, un Intel Xeon de 3 Ghz puede realizar 3 * 4 = 12 Gflop por segundo, o lo que es lo mismo 12 mil millones de operaciones en coma flotante por segundo.

Esto lleva a pensar en el concepto de rendimiento pico y de rendimiento sostenido, puesto que aunque un procesador pueda realizar todo este número de operaciones de

coma flotante por segundo, debe realizar también muchos otros tipos de operaciones: operaciones en memoria, operaciones con enteros, etc.

Desde la década de los 90, se ha impuesto la llamada tiranía de los Flops a través del benchmark Linpack. Muchos de los procesadores y compiladores son optimizados para dar el mayor valor sobre este benchmark para aparecer en la mejor posición en el top500 que recoge los 500 computadores más potentes del mundo ejecutando dicho benchmark.

La característica principal del benchmark de Linpack es que hace un uso muy intensivo de las operaciones de coma flotante. Por ello cuantas más FPU tenga un procesador mejor será su resultado en Linpack.

El 90% del tiempo ejecuta una serie de rutinas BLAS (Subrutinas de Álgebra Lineal Básica), exactamente en la operación DAXPY que es el siguiente cálculo:

y(i) = y(i) + a * x(i).

Por ello, el benchmark Linpack mide prácticamente lo bueno que es un procesador ejecutando operaciones DAXPY.

Por otro lado, al realizar esencialmente cálculos con matrices es un benchmark muy fácilmente paralelizable y se puede usar para comparar sistemas multiprocesadores.

Es ciertamente curiosa la directa relación que existe entre la operación FMA y la operación DAXPI.

Esto, obviamente, es una medida cuanto más subjetiva puesto que lo único que mide este benchmark es cuanto de bueno es ejecutando un tipo de código (DAXPI) que no es significativo con el resto de códigos científicos que existen.

A día de hoy, están apareciendo en escena otro tipo de arquitectura de computadoras denominadas GPU, que utilizan la enorme capacidad de procesamiento de las tarjetas gráficas de tal forma que conjuntando procesadores estándar con tarjetas gráficas, se están diseñando a día de hoy los más potentes computadores del mundo.

Como ejemplo, una tarjeta gráfica en un Tesla S1070 dispone de 240 unidades de coma flotante de precisión simple, cada una de las cuales es capaz de ejecutar 3 operaciones flotantes por ciclo de reloj. Como la frecuencia de cada core de dicha tarjeta es de 1.44 Ghz, dicha tarjeta Tesla C1060 tiene un pico de 1 Tflop por segundo.

4.2.2 TIPO DE PROCESADORES POR SU NIVEL DE PARALELISMO

Se denomina ILP al paralelismo a nivel de instrucción (*Instruction-Level Parallelism*). Bajo este nivel de paralelismo los procesadores se dividen en:

- Escalar (Serie). Cuando sólo puede ejecutar una operación por ciclo de reloj.

- Superescalar: la inmensa mayoría de los procesadores actuales, que pueden realizar varias operaciones diferente a la vez en cada ciclo de reloj.

- *Pipeline*: Comienzan la ejecución de una operación sobre una parte de los datos mientras que está finalizando la misma operación sobre otra parte de los datos.

- *Superpipeline*: Realizar múltiples operaciones de *pipeline* al mismo tiempo.

- Vectorial: Ejecutar la misma operación sobre muchos datos en el mismo ciclo de reloj. Para ello se necesitaba tener registros y unidades aritméticas vectoriales.

Para cálculo científico que suele operar sobre vectores y matrices, tradicionalmente el mejor procesador ha sido el vectorial. Sin embargo, el coste de los sistemas vectoriales ha hecho que prácticamente hayan desaparecido del mercado.

Actualmente, la vectorización ha tomado auge con la aparición de las GPU, puesto que realizan el mismo modelo de operación que los vectoriales a un coste infinitamente más económico.

4.2.3 EJECUCIÓN EN SERIE O EN PARALELO

Lo primero que debe pensar un programador es si va a diseñar su código para ejecutar en serie (en un sólo procesador o núcleo en la tecnología actual) o va a ejecutar en paralelo sobre más de un procesador/núcleo.

El procesador, la memoria y la entrada/salida de datos son los 3 componentes básicos cuando se programa en secuencial, esto es cuando un solo procesador va a ejecutar el código.

Sin embargo, cuando se programa en paralelo además de lo anterior, se debe añadir un cuarto componente básico: la red y la arquitectura de interconexión entre los procesadores (MPP, SMP, cluster, etc.).

En este sentido y tradicionalmente, los programas se han escrito para ser ejecutados en serie, esto es un solo procesador ya que paralelizar requiere de mayores conocimientos de programación.

Desde este punto de vista, computación paralela es el uso simultáneo de los recursos múltiples del sistema de cálculo para solucionar un problema. Los recursos del cálculo pueden incluir:

- Un solo computador con más de un procesador.

- Diferentes computadores conectados por un sistema de red.

- Una combinación de ambos.

Para que un problema pueda ser resuelto mediante computación paralela se debe encontrar tareas que puedan realizarse al mismo tiempo. A tal efecto, existen 2 métodos de descomposición del problema [Cisneros, 2003]:

PARALELISMO FUNCIONAL

- Ventajas:
  - Diferentes procesadores ejecutan diferentes funciones.
  - Procedimiento natural para aquellos programadores formados en programación modular.
- Desventajas:
  - Al crecer el número de procesadores se deben encontrar tareas para todos.
  - Se deben definir funciones cuyo peso en tiempo de ejecución esté balanceado para que se usen de una forma eficiente todos los procesadores simultáneamente.
  - Puede necesitar mucha sincronización y movimiento de datos.

PARALELISMO DE DATOS

- Ventajas:
  - Diferentes procesadores ejecutan la misma operación sobre diferentes partes de los datos.
  - Toman ventajan sobre grandes memorias. En la actualidad son normales configuraciones entre 1 y 4 Gigabytes por procesador. En próximos años, este ratio seguirá aumentando.
  - Se necesita que el dominio del problema pueda ser descompuesto.

Hay varias razones para la utilización de la computación paralela. De ellas, las más utilizadas son la reducción del tiempo de ejecución y la posibilidad de resolver problemas más grandes y complejos por usar memorias grandes.

En la actualidad se pueden clasificar los computadores por la forma de acceso y ejecución sobre los datos en tres grandes familias:

- MISD (Múltiples Instrucciones Simples Datos)
- SIMD (Simples Instrucciones Múltiples Datos)

- MIMD (Múltiple Instrucciones Múltiples Datos). Este nombre sugiere que cada procesador es libre de ejecutar cualquier tipo de instrucción sobre cualquier dato independiente de los otros procesadores.

Atendiendo a este sistema de clasificación, el criterio seguido es el tipo de paralelismo del sistema. En este sentido, existen sistemas en los cuales las instrucciones paralelas que se ejecutan son exactamente las mismas sobre conjuntos de datos distintos, y son conocidos como SIMD (*Single Instruction, Multiple Data*). Estos sistemas, también conocidos como sistemas vectoriales, pueden por ejemplo aplicar la misma operación de suma sobre varios sumandos (32, 64 o 128 es lo habitual) simultáneamente en el tiempo (en el mismo ciclo de reloj). Otro tipo de sistemas son aquellos en los que se efectúan operaciones distintas sobre datos distintos simultáneamente en el tiempo. Estos son denominados MIMD (*Multiple Instruction Multiple Data*). Por ejemplo, un sistema de multiprocesadores escalares realiza operaciones distintas sobre datos distintos en cada procesador y proceso.

En general se puede afirmar que existen ciertos tipos de códigos en los cuales la aplicación de las técnicas de vectorización es muy ventajosa. Estos tipos de códigos han sido denominados como “vectorizables”. Sin embargo, la proporción de códigos de este estilo es bastante reducida, y además con la creciente y constante evolución de los procesadores superescalares que proporcionalmente han evolucionado más deprisa que los vectoriales, su utilización va siendo progresivamente más y más restringida.

Sin embargo, con la aparición de las GPU, los modelos SIMD están cogiendo de nuevo más fuerza.

Otra clasificación se basaría por el acceso a memoria [Zacharov, 2001]:

- Multiprocesadores: Espacio simple de direcciones de memoria y memoria compartida entre todos los nodos.
    - Sistemas UMA, de acceso uniforme a memoria.
        - PVP: *Parallel Vector Processor*, como el Cray T90.
        - SMP: *Symmetric Multi-Processor*: La mayoría de los sistemas actuales.
    - Sistemas NUMA, de acceso no uniforme a memoria en la cual la memoria está distribuida entre todos los nodos aunque todos los nodos ven toda la memoria como global.
        - COMA: *Caché Only Memory Architecture,* como el KSR-1 o el DDM en los cuales la memoria local de cada nodo es usada como caché, en contraste a utilizarlas como memoria principal.

  - CC-NUMA: *Caché Coherent NUMA*, sistema NUMA con coherencia de caché en cada uno de los nodos. Ejemplos son el Origin 2000 y el Altix 3000.

  - NCC-NUMA: *Non-Caché Coherent NUMA*, sistemas NUMA sin coherencia de caché, como el SP3 de IBM o T3D de Cray.

- Multicomputadores: Con diferentes espacios de direcciones de memoria en cada nodo.

  - *Cluster*: Se desarrollará una sección entera sobre como construir un *cluster* actualmente puesto que la mayoría de los sistemas de computación actuales son de este tipo.

  - MPP: *Massively Parallel Processor* como el Cray T3E

Según esta clasificación se tiene en una primera división los sistemas multiprocesadores y los sistemas multicomputadores dependiendo de si se tiene un espacio único de direcciones de memoria en el sistema o no.

Atendiendo al criterio de ubicación de la memoria, se puede clasificar los computadores en sistemas de memoria compartida (o *Shared Memory*) y sistemas de memoria distribuida (o *Distributed Memory*).

Los sistemas de memoria distribuida implican que los datos del usuario están repartidos a través de las distintas "porciones" de la memoria del sistema y que este hecho no es "transparente" al usuario. Esto obliga a que el programa distribuya convenientemente los datos y comunique los cambios en los mismos explícitamente mediante la utilización de librerías de paso de mensajes (PVM o MPI p.e.), y en definitiva, implica un sobrecoste importante en cuanto al desarrollo, facilidad de uso y patología de problemas que conlleva. En contrapartida, estos sistemas permiten generalmente utilizar un número muy elevado de procesadores, puesto que al tratarse de arquitecturas distribuidas la ampliación consiste en simplemente añadir más procesadores y memoria e interconectarlos al sistema mediante algún elemento de interconexión.

Por otra parte, los sistemas de memoria compartida permiten un esquema de programación más simple. La generación de códigos para que se ejecuten en paralelo no implica una recodificación de los mismos en alguna de las librerías de paso de mensajes anteriormente mencionadas sino que especificando unas pocas directivas o simplemente con la utilización de compiladores con opción de autoparalelización es suficiente. En contrapartida este tipo de sistemas no suelen permitir aplicar esta facilidad de uso más que con unos pocos procesadores (decenas como mucho).

Existe una manera de juntar lo mejor de cada filosofía teniendo sistemas de memoria físicamente distribuida y lógicamente compartida. Esta dualidad en cuanto a la distribución de la memoria ha pasado a ser denominada como DSM (*Distributed Shared Memory*). Los sistemas tipo DSM pretenden aunar las mejores características de los

sistemas de memoria compartida (facilidad de uso) y de los de memoria distribuida (escalabilidad) obviando los inconvenientes que presentan ambos.

Atendiendo al criterio de tiempo de acceso a memoria desde los procesadores o latencia, existen dos posibles tipos de sistemas: aquellos en que el acceso a la memoria se produce en un tiempo (latencia) fijo, denominados sistemas UMA (o *Uniform Memory Access*), y aquellos en que los tiempos no son homogéneos sino que varían en función de qué procesador y qué porción de memoria consideramos y que se denominan sistemas NUMA (Non *Uniform Memory Access*).

En principio el hecho de que se pueda fijar de antemano el tiempo de acceso en un único valor parecería que simplifica el mecanismo y por tanto minimiza los posibles problemas que se puedan generar. Sin embargo, es precisamente este hecho el que limita poder escalar el sistema a un número elevado de procesadores. Tratando de relajar esta restricción de que los tiempos no sean uniformes, es posible conseguir sistemas de un orden de magnitud superior en cuanto al número de procesadores.

Lógicamente, gran parte del éxito se basa en que este tiempo no uniforme de acceso a memoria no empeore significativamente respecto al más pequeño que corresponde a los accesos más locales en el sistema. Si el acceso a las partes más lejanas de la memoria implica tiempos de acceso muy superiores, entonces se está limitando la eficiencia que pueda obtenerse en la ejecución de aplicaciones paralelas. Este tipo de aplicaciones precisan acceder a la memoria de los distintos procesos (o hilos de ejecución) que la componen y necesitan periódicamente sincronizar los procesos paralelos antes de iniciar nuevas secciones de la aplicación. Si los tiempos de acceso empeoran significativamente, el acceso a los datos y las sincronizaciones limitan enormemente la eficiencia y por tanto la escalabilidad del sistema.

Entre las máquinas tipo NUMA, existe una característica muy importante derivada del hecho de que existen distintos procesadores con sus respectivas memorias caché. Esto propicia la posibilidad de que exista más de una copia de una misma variable en distintas memorias caché de distintos procesadores que estén trabajando en paralelo sobre datos compartidos. Cuando existen diversas copias hay que garantizar que si cualquiera de ellas es modificada por uno de los procesadores, las demás copias serán actualizadas convenientemente para evitar que trabajen con copias obsoletas de los datos. Esta característica se denomina coherencia de caché.

Los sistemas multiprocesador pueden incluir la lógica necesaria y los protocolos adecuados para garantizar la coherencia de las distintas copias del mismo dato que puedan existir simultáneamente en el sistema. Por este motivo, los sistemas NUMA que presentan esta característica son denominados cc-NUMA (*caché-coherent Non Uniform Memory Access*).

Una forma sencilla de implementar la coherencia de caché consiste en monitorizar el tráfico de las conexiones de acceso a memoria y establecer para cada línea de la caché una señal de estado que indica en qué situación se encuentra. Suele utilizarse un protocolo de implementación tipo "s*noopy" (*fisgón). Esto implica un cierto hardware

específico que mantenga los estados de las líneas de caché según las modificaciones que sufran los datos almacenados en ellas y que se encargue de actualizar las copias de datos que hayan sido modificados por otros procesadores.

Mensajes internos son generados automáticamente para restituir las líneas de caché cuando los datos que contienen son modificados y enviados a los nodos del sistema.

Según el esquema que se emplee para estos mensajes es posible que la escalabilidad del sistema para números elevados de procesadores sea pobre debido al gran número de mensajes que consumen un gran ancho de banda y que además son muy sensibles a la latencia del sistema.

Para solucionar este problema se emplea un esquema sofisticado de coherencia de caché basado en un directorio. En este directorio que figura en cada nodo se guarda una lista para cada línea de caché de los otros nodos que comparten la misma línea. En caso de ser preciso enviar mensajes de invalidación, éstos sólo son enviados a los nodos que comparten la línea (o sea, que figuran en la lista) y no a todos los nodos del sistema, con lo cual el coste de la coherencia de caché no crece con el tamaño del sistema sino con el grado de compartición de las líneas de caché, que es una característica del paralelismo de la aplicación, permitiendo al sistema escalar convenientemente sin que la coherencia de caché constituya un problema para ello.

4.2.4 JERARQUÍA DE MEMORIA

La jerarquía de memoria es una de las partes fundamentales a evaluar en el rendimiento de un código.

Actualmente existe una diferencia importante en la evolución que han tenido los procesadores y la evolución de la jerarquía de memoria. En muchos casos, el procesador está parado esperando a que vengan los datos de memoria.

Es por ello que optimizar el acceso a memoria será vital para obtener un rendimiento óptimo en los códigos.

La jerarquía de memoria está compuesta por orden de rapidez de acceso por los siguientes elementos [Trill, 2000]:

- Registros.
- Caché: en sus diferentes niveles: primaria (L1), secundaria (L2), terciaria (L3).
- Memoria principal (RAM).
- Disco.
- Cinta.

- E incluso en los sistemas HSM, los datos que han sido migrados de disco y cinta, pueden residir en sitios remotos.

Como el acceso a memoria es mucho más lento que el tiempo en realizar una operación (en acceder a memoria puede tomar en muchos casos 100 ciclos de reloj), desde hace tiempo se implementaron mecanismo de mejora de acceso a memoria a través de cachés.

La caché es un tipo de memoria especial de acceso muy rápido ya que puede llegar a residir en el propio chip del procesador pero que es muy costosa (de 100 a 10.000 veces más cara que la memoria RAM por byte).

Los datos en caché pueden ser leídos y almacenados en escasos ciclos de reloj.

La idea básica de la caché hace referencia a los conceptos de:

- Cercanía espacial, esto es que la mayoría de los códigos utilizan bucles en los cuales si se accede al elemento N para operar con él, en la siguiente iteración se necesitará el elemento N+1 para realizar la misma operación (cercanía espacial).

- Cercanía temporal: si se ha utilizado un elemento A, en un periodo corto de tiempo se volverá a utilizar este elemento A.

Los datos en caché se guardan igual que se guardan los elementos en memoria, a través de líneas.

Como ejemplo, el Itanium 2 tiene un acceso de 1 ciclo a caché L1 de datos y de instrucciones, de 5 a 8 ciclos a caché L2 y 12 ciclos a caché L3.

Como las cachés tienen tamaño mucho más reducidos (MB) que las memoria (GB), es necesario implementar buenas técnicas de reemplazo de líneas de caché.

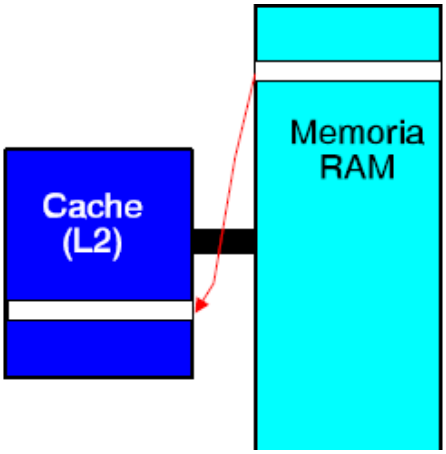

Figura 32: Mapeo de memoria a caché.

4.2.4.1 MAPEO DIRECTO

Consiste en utilizar la siguiente fórmula de reemplazo [De Bustos, 2004]:

Localización de caché = Dirección de Memoria módulo Tamaño de Caché.

Supongamos que el tamaño de caché es de 16 KB = 16 * 1024 Bytes, que corresponde a 4*1024= 4096 palabras de 32 bit.

Se carga un *array* de 32 bit con 8192 elementos.

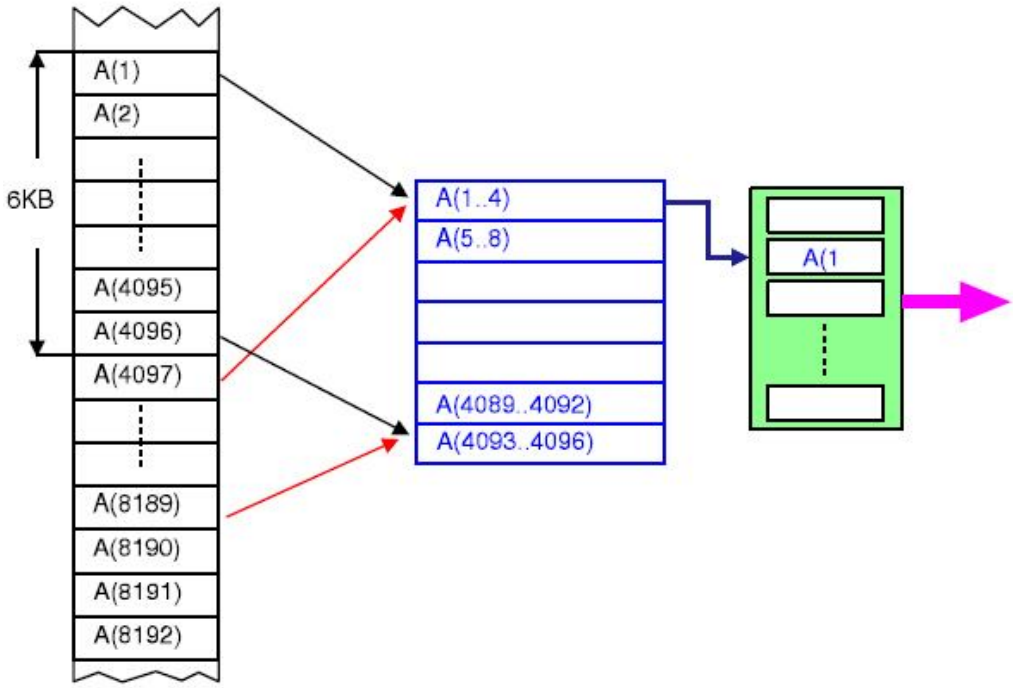

Figura 33: Ejemplo de mapeo directo.

La localización en memoria para el elemento 4097 es igual a:

$$A(4097) = A(1) + 16KB$$

Y la de caché será igual para el elemento A (1) y A (4097).

El problema reside en que se puede producir un patología muy típica que es el "*Trashing*" (trasiego), en el cual los elementos de dos vectores coinciden en el mismo espacio de direcciones y se van reemplazando en cada iteración los datos de uno por los datos del otro elemento, del tal forma que cada referencia a memoria provoca una pérdida de caché.

Sea el siguiente ejemplo:

COMMON A (4096), B(4096)

DO i=1, 4096

PRODUCTO=PRODUCTO+A(i)*B(i)

END DO

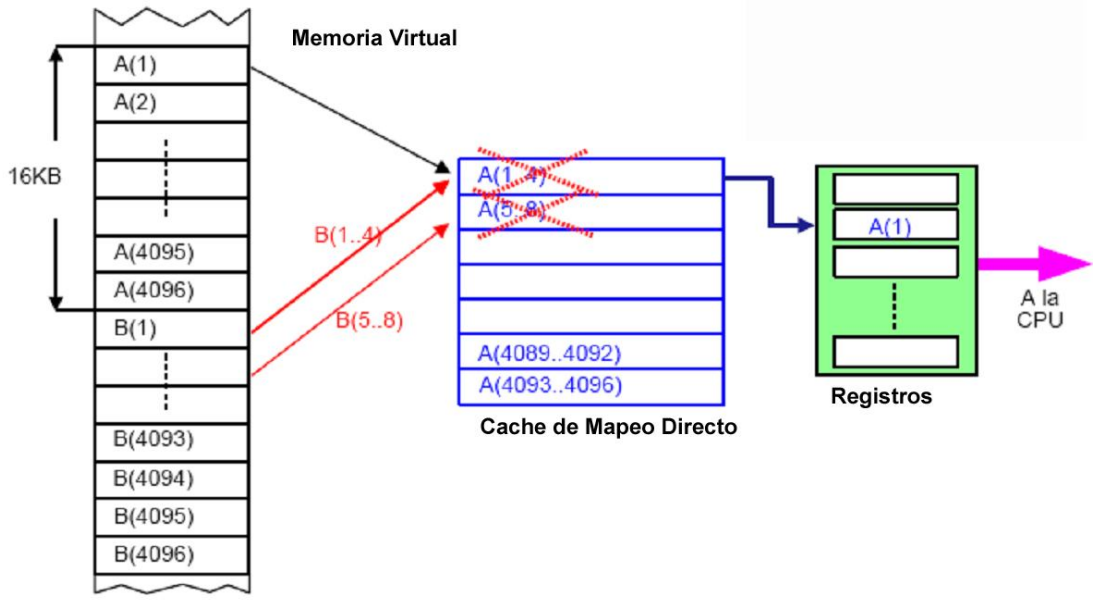

Figura 34: Ejemplo de mapeo directo.

Así los elementos que todavía son necesarios son machacados continuamente. En especial, cuando hay muchos "*arrays*" involucrados este esquema resulta muy ineficiente.

Una solución pasaría por modificar el mapeo a memoria, pero no es algo trivial.

4.2.4.2 CACHÉS FULL ASSOCIATIVE

El algoritmo de reemplazo utilizado es un LRU, "*Least-Recent Use*" (uso menos reciente).

De esta forma, las líneas de caché más antiguas son las reemplazadas.

En la mayoría de los casos esta política funciona muy bien y ayuda mucho con múltiples *arrays*.

No obstante el coste de estas cachés es muy grande, por lo que en la realidad se utiliza una mezcla de las 2 anteriores.

4.2.4.3 N-SET ASSOCIATIVE CACHÉ

La caché contiene varias cachés con mapeo directo a la vez.

De esta forma, las líneas de caché pueden ir a cualquiera de cada una de las subcachés.

La selección de a cual debe ir, se realiza mediante política LRU.

La siguiente figura muestra una caché asociativa de 4 vías (*4-way set associative*)

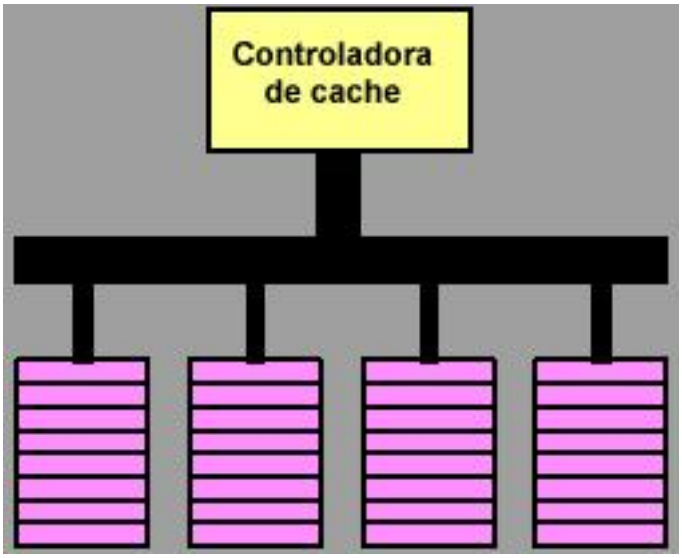

Figura 35: Ejemplo de caché de 4 vías asociativa.

Este tipo de cachés son efectivas en coste, eliminando los problemas de trasiego de las de mapeo directo.

Como ejemplo, en la familia *Happertown* de Intel (familia Intel Xeon 5400), cada procesador tiene una caché L2 de tamaño 6144 KB 24-*way set associative* con 64 byte de línea de caché.

4.2.5 COHERENCIA DE CACHÉ

La coherencia de caché asegura que se dispone del valor correcto independientemente de su localización.

Al existir más de un procesador en el sistema, cada procesador ve toda la memoria total del sistema.

Al pedir un dato a memoria, el procesador va a guardar una copia en la caché local del procesador, por lo que hay que guardar la coherencia de los datos de tal forma que si otro procesador pide esos mismos datos, se asegure que sea coherente.

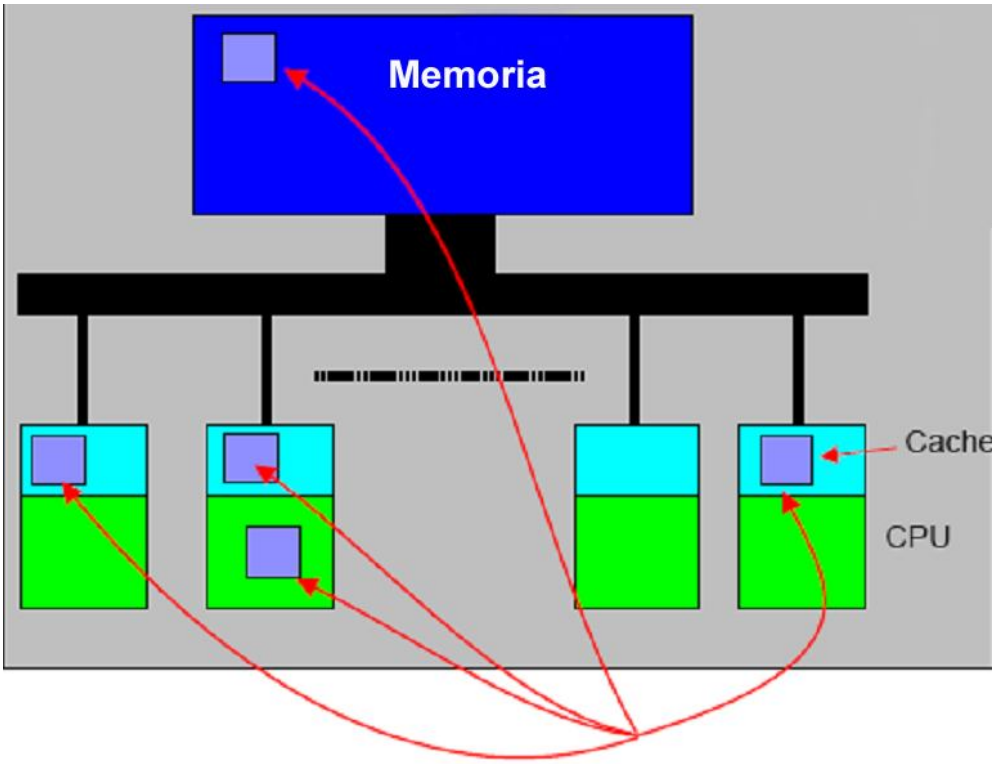

Figura 36: Coherencia de caché.

La misma variable puede estar presente en múltiples localizaciones.

Existen muchas técnicas de coherencia de caché:

- *Write-Through*: Siempre que se escribe en caché, se escribe también en memoria. No tiene muy buen rendimiento pero permite mantener de una forma fácil la coherencia de caché.

- *Write-Back*: Sólo se escribe de la caché a memoria cuando dicha línea de caché es requerida por otro procesador. Da muchas mejores prestaciones que la anterior

- Protocolo *Snoopy*: Todos los procesadores están monitorizando los datos que pasan por el bus para cambiar sus líneas de caché si alguna línea que tiene localmente ha sido cambiado por otro procesador.

- Directorio: Se mantiene en cada línea de memoria unos bits que determinan qué procesador tiene esa línea y en que estado está (limpia, compartida, sucia o inválida). Este tipo de coherencia de caché es el más escalable puesto que sólo aquellos procesadores que tienen que ver con la línea de caché están involucrados.

Sin embargo, es más costoso de implementar. Las cachés basadas en directorios pueden causar en los códigos paralelos una patología muy común que se denomina *False Sharing*, de tal forma que cada uno de los hilos de ejecución o *threads* que están resolviendo un problema en paralelo se están invalidando las líneas de caché mutuamente.

4.2.6 CLUSTER

En la actualidad, la mayoría de los sistemas de cálculo están construidos bajo la arquitectura de *cluster*. Ello es debido a que al buscar el mejor ratio de precio/prestaciones, la arquitectura *cluster* permite aumentar en gran medida este parámetro con respecto al resto de arquitecturas de computación estudiadas.

Un *cluster* consta de los siguientes elementos:

- *Front-end* o nodo frontal: Los usuarios se conectan a un servidor desde el cual compilan y lanzan los programas al gestor de colas. Este nodo hace de entrada a los usuarios y esconde las características y peculiaridades que hay detrás. Para configuraciones con muchos nodos (a partir de treinta y dos nodos de cálculo) es recomendable situar dos nodos que hagan esta función para que en caso de caída del sistema frontal, no se dejen el servicio de entrada al *cluster* cerrado a los usuarios.

- Gestor de Colas: Es un programa que permite organizar y repartir los trabajos de los usuarios a los diferentes nodos. El usuario nunca debe acceder directamente a los nodos. Los gestores de colas permiten repartir la carga de trabajo sobre los diferentes nodos de cómputo de tal forma que se optimice los recursos que se tienen. De esta forma se genera un uso eficiente de todos los recursos. En la actualidad existen muchos tipos de gestores de colas y de recursos como son LSF de Platform, PBS Pro de Altair y de Open Source como Sun Grid Engine, Torque/Maui, Slurm, OpenPBS, etc.

- Servidor de Administración: Es el nodo que permite la gestión de todos los elementos del *cluster*. Desde este servidor se puede realizar el despliegue del sistema operativo sobre todos los nodos de cómputo de una forma rápida, instalación de parches, instalación de software, etc. Sólo debe ser accesible por el administrador y gestiona toda la configuración del cluster. Adicionalmente recibe todas las alarmas y centraliza los mensajes de error de todos los elementos.

- Red de paso de mensajes (MPI). Para aumentar las prestaciones de un código, una de las técnicas posibles es la paralelización. El interfaz MPI (*Message Passing Interface*) es el más utilizado desde hace muchos años. Un *cluster* está orientado a esta forma de programación. Para ello se necesita que los nodos estén conectados entre si por una red de baja latencia y alta velocidad. Actualmente la red que ofrece las mejores prestaciones y es la más difundida es la red de Infiniband, pudiendo ser DDR de 20 Gbit y QDR de 40 Gbit.

- Red de Administración: Es la red por la cual el administrador se puede conectar a todos los elementos del sistema, para realizar tareas típicas de administración como instalar sistema operativo, parar o arrancar componentes, etc.

- Red de Datos: Los nodos deben compartir información y datos entre ellos. La forma más tradicional de realizar esta compartición de datos es a través de una red NFS (*Network File System* o sistema de ficheros en red). No obstante, cuando los requisitos de E/S son exigentes, se recomienda utilizar otras redes de E/S más sofisticadas como puede ser *Lustre*.

- Unión de Redes: Aunque lógicamente en un *cluster* siempre deben existir estas tres redes, para minimizar los costes se suelen unir estas redes en uno o dos redes físicas aunque cada una de ellas tenga direcciones IP diferentes.

- Red Externa: El nodo que hace de frontal estará conectado a la red externa del centro para que los usuarios puedan acceder desde sus estaciones remotas. El nodo de administración también está conectado a esta red externa.

- Espacio Temporal o *Scratch*: Es el espacio temporal que se usa para la ejecución del programa. Se necesita que sea rápido, bien con discos locales o bien a través de usa solución de sistema de ficheros compartidos como puede ser lustre, GPFS, SFS, ZFS, etc. NFS no es aconsejable como sistema temporal puesto que las prestaciones que da son muy bajas actualmente aunque hay versiones de NFS como *Parallel NFS* que resuelve este problema.

- Cuentas de usuario o *Home*: Los usuarios guardan sus datos de entrada y de ejecución en un espacio no temporal que debe perdurar y sobre el cual se debe hacer copia de seguridad.

- LCD: Se suele utilizar un monitor con teclado y ratón en el armario para mayor comodidad en caso de perder el acceso al servidor de administración o perder la red de administración por una avería.

A continuación se muestra un esquema con todos los componentes:

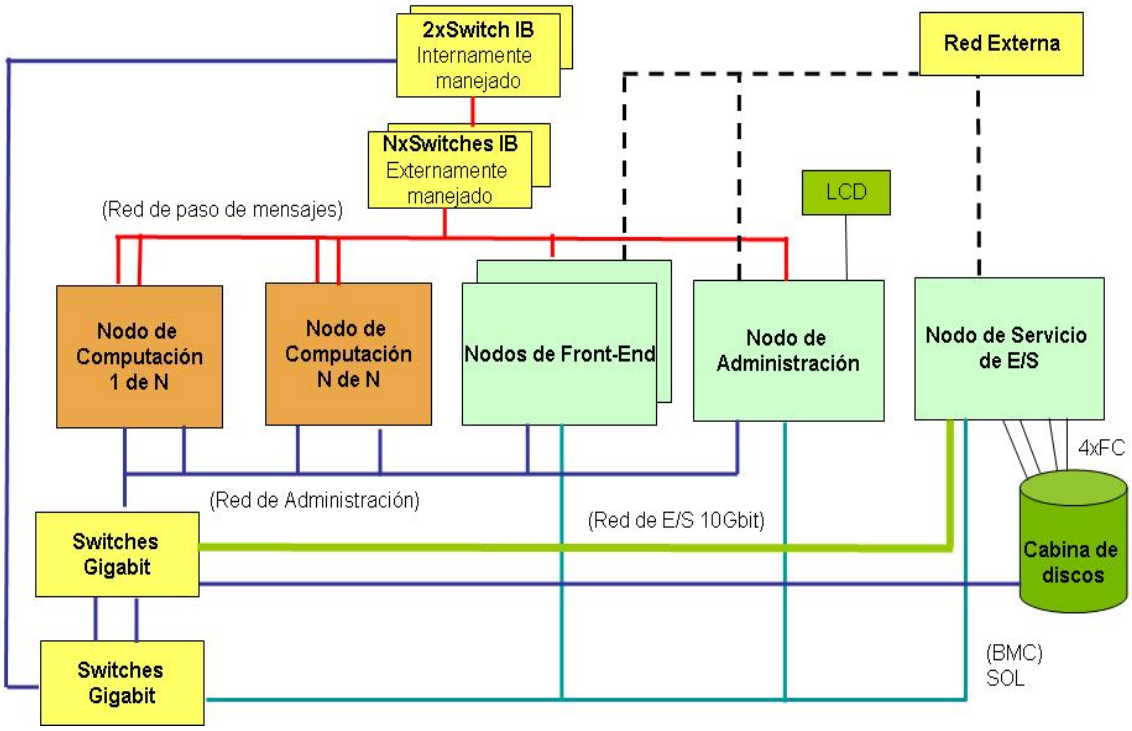

Figura 37: Arquitectura de un *cluster*.

4.2.7 ARQUITECTURA GPGPU

En la actualidad existen diferentes opciones de acelerar los códigos científicos utilizando tarjetas gráficas para cómputo.

- Tarjetas Tesla de NVIDIA.
- Tarjeta FireStream de ATI.
- Tarjeta Cell de IBM.

No obstante, los dos proveedores dominantes actualmente son NVIDIA e IBM, ya que ATI está retrasado con respectos a sus dos competidores.

En las tres, es necesaria la conexión de la tarjeta a un servidor de propósito general.

A modo de ejemplo se profundizará un poco más en la arquitectura Tesla de NVIDIA.

4.2.7.1 ARQUITECTURA TESLA DE NVIDIA

En Junio del 2008, NVIDIA presentó la segunda generación de arquitecturas de computación paralela con la aparición de la tarjeta GT200 [Nvidia, 2008].

Esta tarjeta tiene 1.4 billones de transistores, con un total de 240 unidades de coma flotante para un rendimiento pico de cerca de 1 Tflop (933 Gflops).

Para el mercado de computación ha sido desarrollado bajo el nombre de T10.

Cada tarjeta T10 contiene hilos de ejecución denominados *Thread Processors* (TP) que contiene registros y unidades de procesamiento entero y flotante.

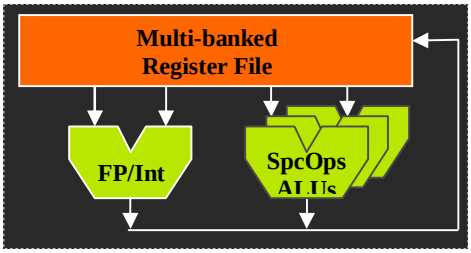

Figura 38: Esquema de un TP.

Los TP se agrupan formando *Thread Processor Array* (TPA) que contienen:

- 8 unidades TP.
- Memoria compartida.
- 1 unidad de doble precisión.
- 1 unidad de función especial.

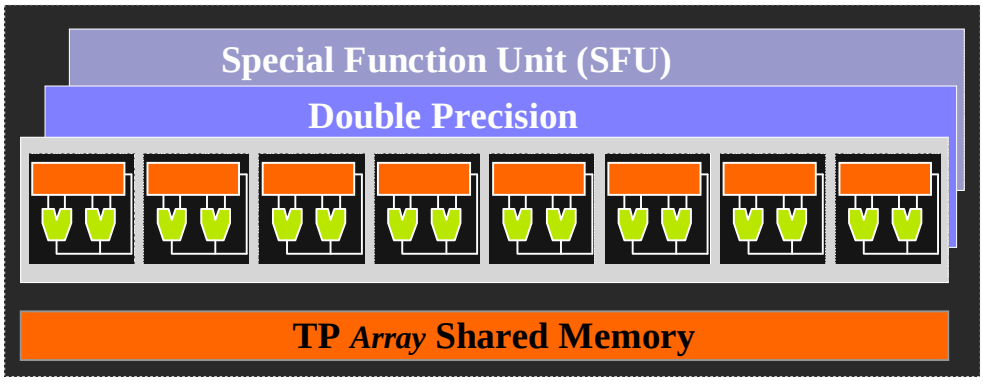

Figura 39: Esquema de un TPA.

En una tarjeta NVIDIA T10P se agrupan finalmente 30 unidades TPA con las siguientes características principales:

- 240 unidades de coma flotante de precisión simple.
- 30 unidades de coma flotante de precisión doble.
- 4 GB de memoria GDDR3 con un ancho de banda de memoria de 102 GB/s.

Para manejar estos cientos de hilos de ejecución, el multiprocesador emplea una nueva arquitectura denominada SIMT (*Single Instruction Múltiple Thread*).

El multiprocesador mapea cada hilo de ejecución sobre una unidad de procesamiento escalar y cada unidad se ejecuta de forma independiente con su propia dirección de instrucciones y estado de registro.

La arquitectura SIMT es muy parecida a la arquitectura vectorial SIMD (*Single Instruction, Multiple Data*) en la que una instrucción simple controla el procesamiento de múltiples elementos. La diferencia clave con una organización vectorial SIMD reside en que la arquitectura SIMT permite a los programadores escribir código independiente paralelo a nivel de *thread* así como código paralelo a nivel de datos para coordinar dichos *threads*.

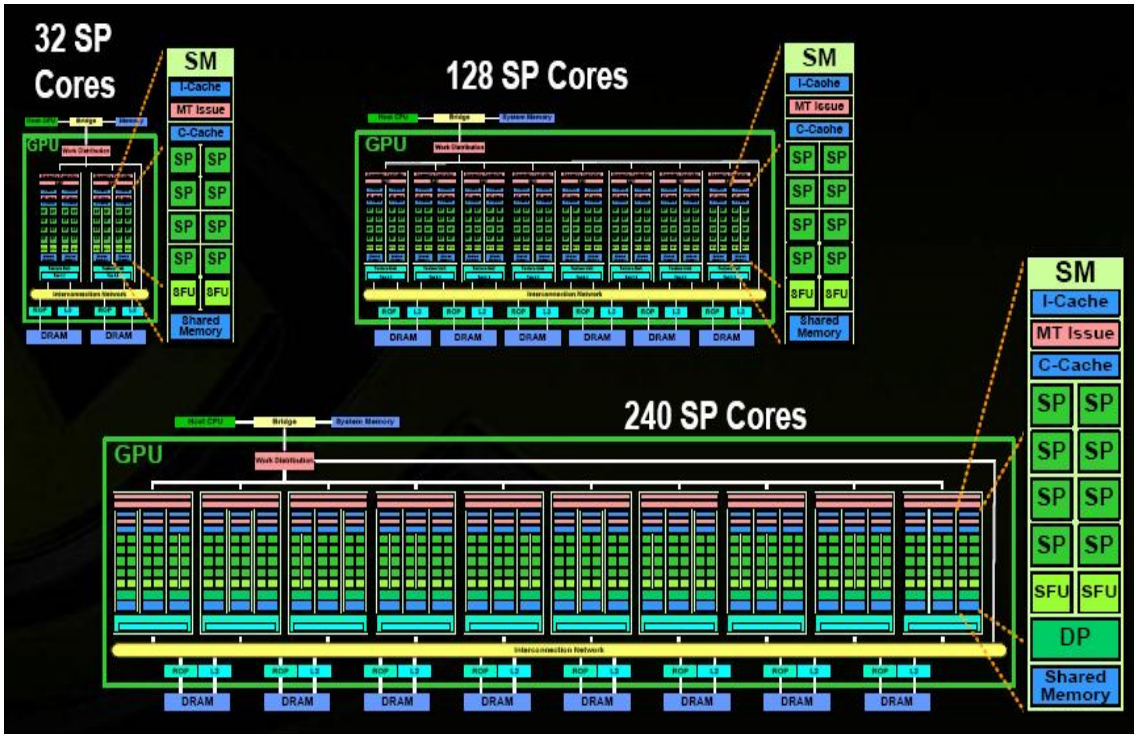

Figura 40: Diferencia de arquitectura de la generación 8 a la 10 de NVIDIA.

En esta figura se aprecia la diferencia de arquitectura entre la generación 8 y la 10 de NVIDIA.

La tarjeta gráfica T10P se comercializa con el nombre Tesla C1060 y debe ser conectada a un *slot* PCIe x16 Gen2 para conseguir el máximo ancho de banda a memoria (5.5 GB/sg)

Figura 41: Tarjeta Nvidia C1060.

No obstante, se puede conectar 4 GPU en un único servidor formando un Tesla S1070.

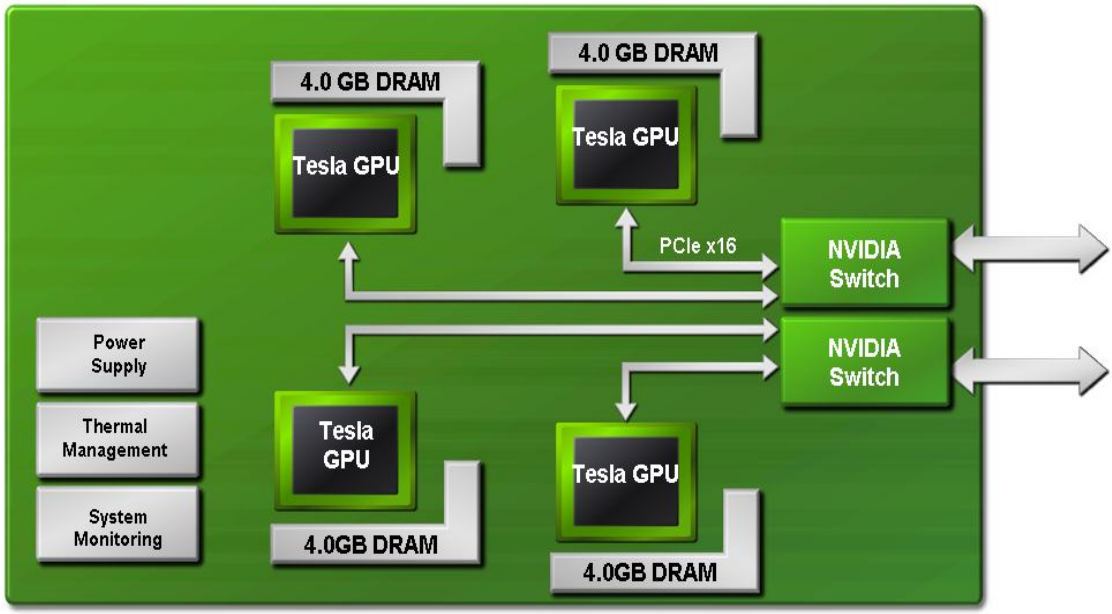

Figura 42: Servidor NVIDIA Tesla S1070.

De esta forma, mediante cables PCIe x16 se conecta este servidor a un servidor de propósito general:

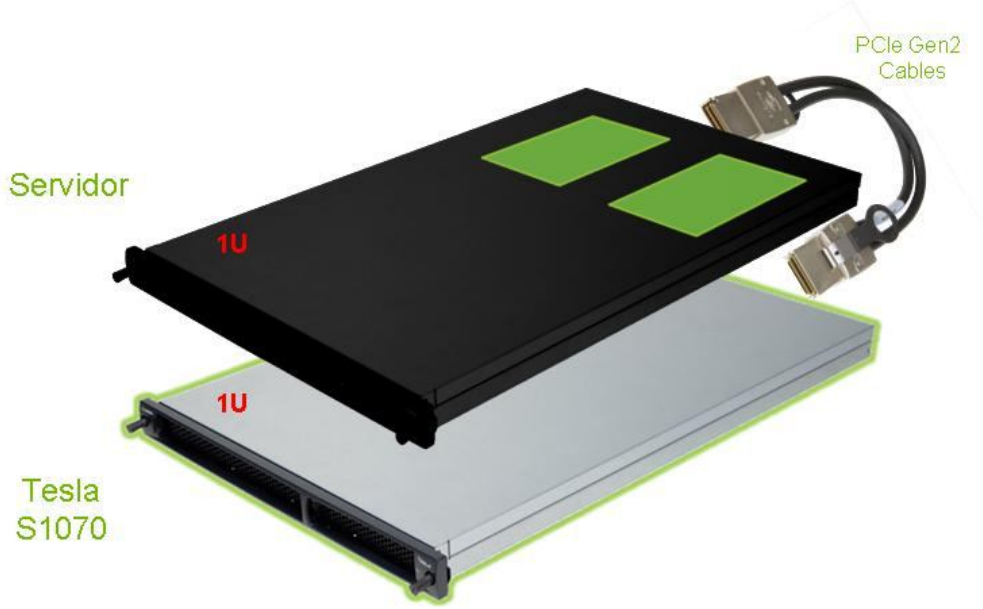

Figura 43: Servidor NVIDIA Tesla S1070.

Pudiéndose conectar también un mismo servidor Tesla S1070 a dos servidores de propósito general ya que internamente el servidor Tesla S1070 está separado como se ha visto anteriormente por 2 switches:

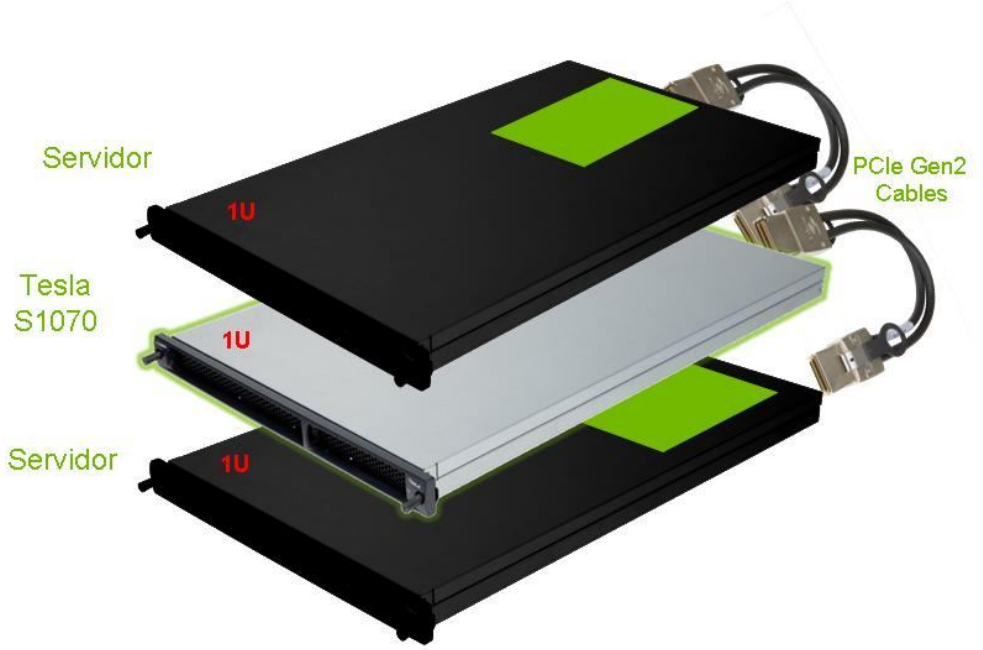

Figura 44: Conexión de un servidor NVIDIA Tesla a un host.

Sea cual sea la conexión, lo que se consigue es que aquellas operaciones matemáticas (multiplicación de matrices, transformadas, etc.) que necesiten ser aceleradas, se transmitan del servidor de propósito general al servidor Tesla.

Los *speedup* que se están obteniendo con esta nueva arquitectura son extraordinarios desde un 10x hasta 200x dependiendo del código por lo que están revolucionando los códigos científicos.

5 PROCESO DE OPTIMIZACIÓN DEL RENDIMIENTO

5.1 INTRODUCCIÓN

El proceso de optimización de códigos científicos empieza por partir de un código ya creado anteriormente o por la creación de dicho código desde cero.

Si el código se va a escribir desde cero, se pueden plantear una serie de fases que son difíciles de realizar cuando el código ya está escrito anteriormente.

La mayoría de los programas no comerciales utilizados actualmente están escritos en Fortran o en C y se dispone del código fuente. Sobre esta última división se podrá implementar una metodología de optimización puesto que sobre los códigos comerciales no se puede implementar técnicas de mejora de rendimiento al no disponer en la mayoría de los casos del código fuente original.

En todo caso, la metodología que se va a mostrar necesitará de herramientas adicionales como son las herramientas de *profiling* y de *debugging* para localizar las patologías dentro del código.

Estas herramientas en la mayoría de los casos son comerciales aunque se pueden encontrar también alguno con buenas características de código abierto.

5.2 PROCESO DE OPTIMIZACIÓN

La metodología a utilizar será la siguiente:

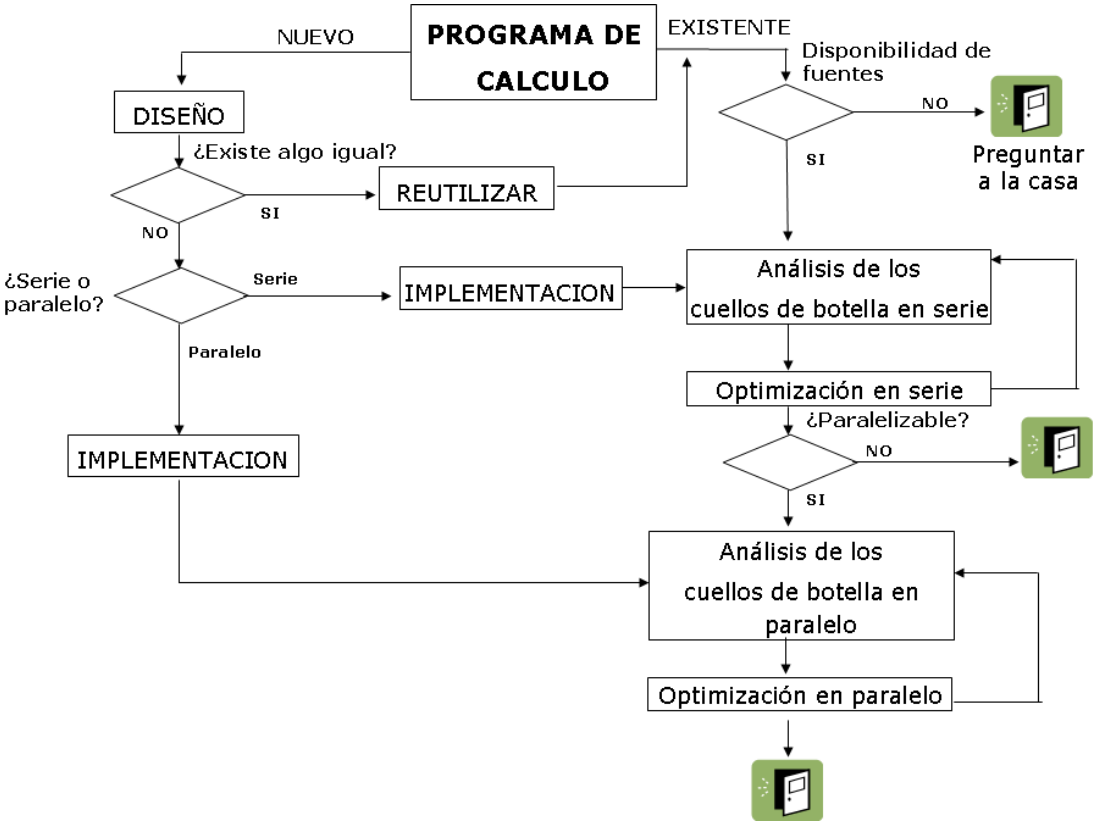

Figura 45: Metodología de optimización de códigos.

1. Determinar si se va a utilizar un código ya creado o bien se partirá de cero.

2. Tanto en el caso de crear un código propio como de disponer del código fuente de uno ya creado, la metodología presenta una serie de etapas comunes:

- Análisis de los cuellos de botella en serie.
- Optimización del código en serie.
- Análisis de los cuellos de botella en paralelo.
- Optimización del código en paralelo.

Este proyecto de fin de carrera se ha centrado exclusivamente en el análisis de los cuellos de botella y optimización del código en serie, dejando para futuros desarrollos que pudieran ser la continuación de este proyecto de carrera con una tesis doctoral, la optimización de código en paralelo o su migración a GPU.

6 ANÁLISIS INICIAL DE UN PROGRAMA DE CÁLCULO

Antes de escribir un programa de cálculo, se debe hacer un análisis previo para comprobar si existe algo parecido o igual a lo que queremos implementar.

Este análisis permite ahorrar un gran tiempo de desarrollo para no volver a escribir algo que ya esté hecho anteriormente.

En la actualidad, los científicos suelen utilizar o bien códigos comerciales o bien sus propios códigos.

Se ha dividido el software científico por disciplinas para una mejor búsqueda para el lector:

- CSM (*Computational Structural Mechanics*): Abaqus, Ansys, MSC Marc, LS-Dyna, Pam-crash, Radioss, Madymn, Adina, AMSL, Optistruct, Nastran, Permas, SAMCEF, Sysnoise, MSC Dytran, Autoform.

- CFD (*Computational Fluid Dynamics*): Fluent, Star-CD, Fire, CFX, PowerFlow, Vectis, CFD-Fastran, GASP, GT-Power, PAM-FLOW, STARCCM+, CFD++, ACUSOLVE, WAVE, MoldFlow, SWIFT, Phoenics, FOAM, USAero, VSAero, USAero, MGaem, CFD-ACE, Radioss-CFD

- CEM: (*Computational Electromagnetics*): FMSlih, FEKO, Xpatch, MM3D, PAM-CEM, HESS, .IMAG, XFDTD, SIBLBC.

- CCM (*Computational Chemistry and Material Science*): Amber, Gaussian, ADF, Gamess, NAMD, NWChem, VASP, ADF, Band, Charmm, Siesta, Gromacs, Molpro, WIEN2k, Castep, DMLO3, CCP4, Discover (MS3), Mesodyn(MS3),VAMP(MS3), CPMD, CP90, Jaguar, Paratec, Turbomole, Q-Chem, Dock, PWscf, Mopac, Solve/Resolve.

- Bioquímica: HMMER, BLAST, FASTA, ClustalW, NCBI-BLAST, Emboss, BioFacet, SPS Phrap, SPS Swat, SPS Pfam, Genewise, CAP3, Mascot, SRS, Phylin.

- Procesos sísmicos y de reservas: Eclipse, VIP, PrnMax, GeoDepth/EPOS, Omega, SeisUp, Visage, GIGAViz, GeoCluster, Stratimagic, Focus/EPOS, ComAz.

- Simulación del clima, tiempo, océanos: WRF, MM5, POP, CCSM, ECHAM, Aladin, MOM4, IFS, HRM, GFS, NFS.

En muchos de los casos, son códigos científicos sobre los cuales hay que adquirir licencia para su uso.

Otros, no obstante, tienen licenciamiento libre.

En ambos casos, tanto si es libre como si la licencia es comercial, los creadores permiten disponer de los códigos fuentes para su uso.

En caso de no disponer del código fuente, la mejor opción será preguntar a los creadores de la aplicación sobre cual es la mejor arquitectura para utilizar el software.

En caso de no disponer de un programa que pueda servirnos como base y se decida implementar un código científico nuevo, se debe plantear en el diseño si se desarrollará el código para ser ejecutado en paralelo o en serie, dado que desde su creación será más fácil diseñarlo para ser ejecutado en paralelo que tratar de paralelizarlo posteriormente.

7 ANÁLISIS DE LOS CUELLOS DE BOTELLA DE UN CÓDIGO SERIE

El proceso de optimización de un código es un proceso iterativo y cíclico como se muestra a continuación:

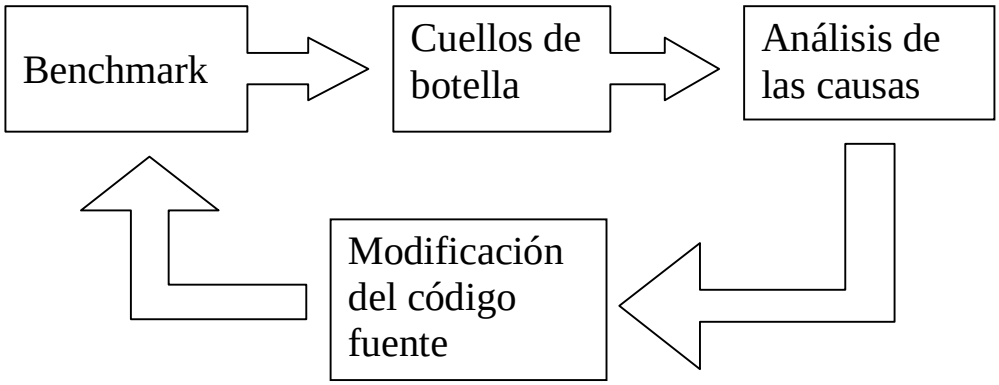

Figura 46: Proceso de optimización de un código monoprocesador.

7.1 BENCHMARK

Dependiendo de los juegos de entrada de un programa, seguramente se ejecutarán una serie de funciones diferentes en cada caso.

Es por ello que a la hora de optimizar un código, se debe analizar previamente que parte del código se va utilizar y para ello se creará un juego de datos ficticios o reales que nos permitirá analizar el rendimiento del mismo con estos datos de entrada. Este proceso se denomina *benchmarking*.

Un *benchmark* generalmente toma menos tiempo que un caso real para que el proceso de optimización pueda llevarse a cabo sin tener que perder mucho tiempo en la ejecución del caso.

7.2 CUELLOS DE BOTELLA

La localización de cuellos de botella en el código suele ser realizada con herramientas disponibles en el mercado.

Básicamente lo que hacen estas herramientas es instrumentar el código para localizar aquellas zonas en las cuales se gasta más tiempo de proceso, más tiempo de acceso a memoria, mayor número de fallos de caché, etc.

Son muchas las patologías que puede sufrir un código y este tipo de herramientas nos sirven de base para encontrar este tipo de errores.

En el mercado hay muchas herramientas de análisis de prestaciones, algunas comerciales como *Intel Vtune* y otras de libre distribución como el *gprof* o el *pfmon*.

Estas herramientas nos van a permitir encontrar las zonas del código en las cuales se está consumiendo o bien más tiempo, o bien está mostrando una anomalía: muchas pérdidas de caché, mucho acceso a memoria, mucha operaciones de entrada/salida, etc.

7.3 ANÁLISIS DE LAS CAUSAS Y MODIFICACIÓN DEL CÓDIGO

Una vez localizadas las funciones que están causando los cuellos de botella, se debe analizar las causas de estas anomalías y la modificación del código para evitar estas anomalías.

Estas zonas son debidas o bien a funciones que se repiten muchas veces en la ejecución del código o bien a funciones que toman mucho tiempo en ejecutarse.

Esta es quizás la parte más difícil del proceso de optimización puesto que requiere un conocimiento amplio sobre arquitectura de computadores y de programación.

Es por ello, que la parte principal de este proyecto fin de carrera se centra en esta parte.

8 OPTIMIZACIÓN EN SERIE

8.1 INTRODUCCIÓN

Para optimizar un código en serie, hay que analizar previamente con alguna herramienta que tipo de patologías está detectando.

Como hemos visto anteriormente, una vez encontrado el origen, habrá que modificar el código para evitar esta patología.

Las causas principales que ocasionan que un código se ejecute a menor rendimiento del esperado son debido a problemas con:

- Utilización de un lenguaje de programación equivocado o mal usado.
- Algorítmica.
- Utilización de librerías y código ya optimizado.
- Optimización de bucles.
- Optimización interprocedural.
- Acceso a memoria: caché, memoria principal y entrada/salida.
- Otras causas.

Es por ello que en este TFC se analizará como solucionar problemas en cada uno de estos puntos.

8.2 UTILIZACIÓN DE UN LENGUAJE DE PROGRAMACIÓN CORRECTO

Existen numerosos lenguajes de programación disponibles en el mercado. Hasta la fecha se puede hablar de miles de lenguajes de programación de todo tipo (funcionales, procedurales, lógicos, orientados a objetos, etc.) aunque sólo unas docenas de ellos son utilizados actualmente de una forma extendida.

Entre todos ellos se puede hablar de los históricos Fortran, C, Cobol, Basic, Prolog, Pascal, C++, Algol, Lisp que han perdurado hasta día de hoy y han dado en algunos casos origen a otros más recientes (VB.NET, Java, Php, Perl, etc.).

La conocida editorial O'Reilly cita más de 2500 lenguajes de programación. El primer lenguaje nació en 1954 y fue Fortran, el cual permanece hasta hoy en día siendo el mejor lenguaje científico hasta la fecha.

De todos ellos, dependiendo de los objetivos iniciales que se tengan a la hora de plasmar las ideas, puede resultar más cómodo usar un lenguaje u otro. Muchas veces viene dado también por el hecho de poder reutilizar código con funciones matemáticas escritas por alguien previamente, y lo más importante, corregidas, validadas y eficientes.

Como se indicaba en un capítulo previo, dependiendo de la arquitectura que tengamos y del compilador que estemos utilizando, el código generado será más eficiente utilizando un lenguaje u otro.

En la experiencia, aquellos códigos escritos sobre Fortran suelen ser los más eficientes puesto que el compilador suele interpretar de una forma más o menos fácil y eficiente lo que el programador pretende hacer.

Conforme los lenguajes de programación aumentaron su complejidad y funciones, los compiladores a su vez aumentaron su complejidad y a pesar que cada vez son más eficientes, la experiencia me ha demostrado que Fortran sigue siendo el lenguaje de programación más eficiente para implantar códigos científicos.

Esto es debido a que con la aparición lenguajes en los cuales se podía utilizar punteros y otras técnicas de acceso a variables en memoria, simplificó la vida a los programadores a expensas de crear códigos menos eficientes.

Un código escrito en C o C++ puede llegar a ejecutarse en el doble de tiempo con respecto al mismo código escrito en Fortran 77 o 90.

En general, basándome en mi experiencia analizando códigos de clientes, este factor ha ido desde un valor 1 (esto es, no se apreciaba prácticamente diferencia de rendimiento) hasta incluso un factor 3. Todo ello depende de la forma en que se haya escrito el código en uno u otro lenguaje y lógicamente del compilador utilizado.

Otros lenguajes de carácter multiplataforma como puede ser Java, han dado rendimientos que se pueden considerar muy malos para códigos científicos. Esto es debido a que Java necesita interpretar el código en tiempo de ejecución, siendo claramente un lenguaje poco orientado a rendimiento. A pesar de ello, su filosofía multiplataforma la hace ser muy interesante para otros tipos de programas.

Otro punto importante a la hora de elección del software, es el sistema operativo a utilizar. Actualmente los sistemas estilo UNIX como puede ser Linux, están dando las

mejores prestaciones a la hora de ejecutar un código. Además, la mayoría de los científicos desarrollan los códigos sobre este sistema operativo, por lo que desde mi punto de vista, la mejor selección de sistema operativo es actualmente Linux.

Algunas guías generales para escribir un código eficiente son las siguientes:

- Uso de variables locales, preferentemente automáticas en vez de globales. El compilador puede analizar con exactitud el uso de las variables locales, pero no puede realizar tales asunciones cuando se utilizan variables globales por que lo que debe inhibir ciertas optimizaciones.

- Evitar mezclar diferentes tipos de datos en expresiones aritméticas (sumas, restas, multiplicaciones, etc.) ya que puede causar conversiones de tipo en tiempo de ejecución entre enteros o flotantes lo que degradará el rendimiento. De hecho, tampoco es recomendable mezclar el mismo tipo de datos pero con diferentes precisiones ya que producirán también degradaciones en el redondeo.

- Usar la precisión de coma flotante más pequeña posible en general. Las operaciones de coma flotante en precisión simple suelen ser más rápidas que las de doble precisión excepto si el procesador realiza todas sus operaciones en doble precisión (como es el caso del procesador Power de IBM). Sin embargo para el caso de IBM POWER, todas las operaciones de coma flotantes se realizan en doble precisión. Cuando ha de realizar una operación en precisión simple, el hardware convierte todos los operandos a doble precisión, ejecuta la operación, y posteriormente redondea el resultado convirtiendo el resultado nuevamente a precisión simple. Estas conversiones tienen un impacto negativo en el rendimiento.

- En Fortran, evitar usar *EQUIVALENCE statements*. El uso de *EQUIVALENCE statements* puede provocar datos no alineados (*unaligned data*) o causar que los datos se salgan de los límites naturales definidos. Además puede evitar ciertas optimizaciones muy positivas como el análisis de datos globales o *implied-DO loop collapsing* cuando las variables de control están en un *EQUIVALENCE statement.*

- En Fortran 90, usar variables *module* más que *common blocks* para almacenamiento global. Usar módulos ayuda a asegurar la consistencia de alineamientos.

- Evitar el uso innecesario de punteros en C y C++. El compilador tiene dificultades en las optimizaciones con punteros. El uso de punteros inhibe optimizaciones tales como *dead store elimination* y *store motion.*

- Usar Fortran 90/95 *array intrinsic procedures* más que crear rutinas propias.

- En vez de usar bucles explícitos para acceder a *array*, usar la sintaxis de *array* de Fortran 90/95 siempre que sea posible.

8.3 ALGORÍTMICA

La algorítmica es una ciencia por la cual se pretende desarrollar códigos que utilicen los métodos más eficientes para implementar la especificación del problema.

Escoger un algoritmo apropiado es uno de los factores más importantes a la hora de implementar un problema. La algorítmica se basa en que todos los códigos tienen un orden de complejidad y cuanto menor complejidad tenga nuestro código más rápido se ejecutará en cualquier plataforma.

$O(1) < O(\log 2 (n)) < O(n) < O(n * \log 2 (n)) < O(n^2) < ...$

El orden de complejidad lo determina el número de bucles que haya en el código. Ejemplos de orden de complejidad son los siguientes:

O(1):

```
Total = Sum1 + Sum2
```

O(n):

```
DO i= 1, N
        Total = Total + Suma(i)
END DO
```

$O(N^2)$:

```
DO i= 1, N
        DO j= 1, N
                Total = Total + Suma( i, j)
        END DO
END DO
```

Se va a desarrollar este ejemplo con más profundidad para calcular su complejidad exacta:

Cuando i=1, j=1, 2,... N => N iteraciones.

Cuando i=2, j=2, 3,... N => N-1 iteraciones.

Cuando I=3, j=3, 4,... N => N-2 iteraciones.

...

Cuando i= N-2, j= N-2, N-1, N => 3 iteraciones.

Cuando i= N-1, j= N-1, N => 2 iteraciones.

Cuando i= N, j = N => 1 iteración.

Número total de iteraciones: = Si (i = [1.. N])

Total = 1 + 2 + 3 + ... + (N-2) + (N-1) + N

Total = (N+ 1) + ((N-1) +2) + ((N-2) + 3) + ...

Total = (N+ 1) + (N+ 1) + (N+ 1) + ...

Total = (N+ 1) * N/ 2

Total = (N ² + N) / 2

Total = N ² / 2 + N / 2

Este código tendría complejidad por lo tanto de O ($N^2 / 2 + N / 2$) aunque generalmente se considera simplemente el orden de magnitud mayor, que en este caso sería O (N^2) si el número de iteraciones es elevado.

Sin embargo, para un número de iteraciones pequeños los coeficientes (partido por 2, + N, -K, etc.) juegan un papel importante.

Cuando el número de iteraciones es lo suficientemente grande, el peso de las constantes y otros términos apenas afectan en el tiempo de ejecución por lo que se consideran despreciables.

Es fundamental tratar de emplear un algoritmo eficiente que resuelva el problema, puesto que de esta forma ahorraremos mucho esfuerzo en tratar de optimizar nuestro código con otras técnicas que se verán a continuación. Si de partida nuestro algoritmo es el más eficiente posible, nos permitirá posteriormente centrarnos simplemente en aspectos de la arquitectura en la cual estemos ejecutando el código.

Un ejemplo de ello sería el archiconocido problema de ordenación. Para este problema se tienen algoritmos que lo resuelven desde una complejidad logarítmicas (*quicksort*) hasta una complejidad cuadrática (burbuja). Si se escoge el algoritmo de ordenación de complejidad logarítmica se sabrá a ciencia cierta que se está utilizando la técnica más eficiente sobre cualquier computador. De hecho lo más probable es que tardará menos tiempo en ejecutarse en un computador antiguo con el algoritmo de complejidad logarítmica que si se utiliza el computador más moderno y se emplea un algoritmo cuadrático.

8.4 UTILIZACIÓN DE CÓDIGO YA OPTIMIZADO

Una de las facetas más importantes a la hora de escribir un código eficiente es no hacer 2 veces el trabajo que alguien probablemente ya haya realizado alguna vez. Todas las rutinas y funciones matemáticas normales están disponibles en librerías matemáticas de código abierto o propietarias. Por ello, siempre es recomendable perder algo de tiempo buscando alguna rutina que ayude a implementar el cálculo que tratar de escribirla desde cero: va a quedar menos eficiente y con algún error o caso no testeado seguro. Muchos de esas rutinas y fórmulas matemáticas están incluso escritas en código máquina y alguna de las compañías que las escriben (NAG, Intel MKL, SCSL, IMSL, Numerical Recipes, etc) hacen negocio de ello.

La técnica de utilización de código ya optimizado consiste en emplear rutinas de librerías comerciales o no que cumplan los requisitos del problema que se plantee.

Existen en la actualidad numerosas librerías científicas que pueden ser empleadas para este fin. A continuación se va a enumerar las más importantes.

8.4.1 LIBRERÍA BLAS

La librerías BLAS (*Basic Linear Algebra Subprograms*) proporcionan la básica de las operaciones con matrices y vectores para la resolución de la mayoría de las funciones de algebra lineal. Contienen varias rutinas BLAS, incluyendo:

- BLAS Nivel 1 - operaciones vector-vector.
- BLAS Nivel 2 - operaciones matriz-vector.
- BLAS Nivel 3 - operaciones matriz-matriz.
- *Sparse* BLAS: Una extensión de BLAS de los 3 niveles para matrices sparse.

Fueron desarrolladas en la Universidad de Tennessee y son un estándar de la industria.

Información detallada sobre la librería BLAS se puede encontrar en la página web:

http://www.netlib.org/blas/index.html

Existe adicionalmente una versión paralela de BLAS (PBLAS o *Parallel BLAS*) y un conjunto de BLACS (*BASIC LINEAR ALGEBRA COMMUNICATION*) para pasos de mensajes entre procesadores.

8.4.2 LIBRERÍA FFT

Conjunto de rutinas que realizan transformadas *Fast Fourier* así como rutinas lineales como convolución y correlación. Incluyen:

- Raíces mezcladas de 1, 2 y 3 dimensiones.
- Raíces mezcladas múltiples unidimensionales.
- Convoluciones de 1 y 2 dimensiones.
- Convoluciones múltiples de 1 dimensión.
- Correlaciones de 1 y 2 dimensiones.
- Correlaciones de 1 dimensión.
- Precisión simple y doble, para números reales y complejos.
- Soluciones directas para sistemas de ecuaciones lineales dispersas con estructura simétrica no-cero.
- Soluciones interactivas para ecuaciones lineales dispersas.

8.4.3 LIBRERÍA LAPACK

Es una librería de subrutinas para resolver la mayoría de los problemas comunes de álgebra lineal: sistemas de ecuaciones lineales, problemas eigenvalue, y problemas de valor simple. Incluyen:

- Sistemas de ecuaciones simétricos y asimétricos.
- *Eigenvector/value* simétricos y asimétricos.
- *Singular Value Decomposition* (SVD).
- *Linear Least Squares.*

La librería LAPACK fue desarrollada en la Universidad de Tennessee y es también un estándar de la industria.

Información más detallada puede encontrarse en la página web:

http://www.netlib.org/lapack/index.html

8.4.4 LIBRERÍA SCALAPACK

La librería ScaLAPACK (*Scalable Linear Algebra Package*) incluye rutinas para resolver problemas de algebra lineal en sistemas de memoria distribuida (*cluster*). Es un subconjunto de las LAPACK diseñado para ser ejecutado sobre cluster para correr en paralelo. Para más información:

http://www.netlib.org/scalapack/

8.4.5 LIBRERÍA PARDISO

La librería PARDISO incluye funciones paralelas para resolver ecuaciones lineales simétricas y no simétricas grandes que tengan matrices *spare* (aquellas matrices dominadas por ceros en su mayoría).

http://www.pardiso-project.org/

Estas matrices al contener muchos ceros, pueden usar técnicas de compresión para utilizar poca memoria, a la vez que grandes matrices de este estilo, nunca podrían ser resueltos por métodos de resolución de matrices estándar.

8.4.6 LIBRERÍA MATEMÁTICA DE VECTORES

Las VML (*Vector Math Library*) incluyen funciones matemáticas altamente optimizadas sobre C y sobre Fortran.

Aritmética	**Trigonometría**	**Hiperbólicas**	**Potencia y raíces**
Add	Sin^	Sinh^	Pow^
Sub	Cos^	Cosh^	Powx^
Div	SinCos	Tanh^	Pow2 o 3
Sqr	CIS^^	Asinh^	Pow3 o 2
Mul	Tan^	Acosh^	Sqrt^

Conj^^	Asin^	Atanh^	Cbrt
MulByConj^^	Acos^		InvSqrt
Abs	Atan^		InvCbrt
	Atan2		Hypot
			Inv

Figura 47: Funciones matemáticas incluidas en VML.

Redondeo	**Exponenciales/ Logarítmica**	**Especiales**	**Otras**
Floor	Exp^	Erf	Inv
Ceil	Expm1	Erfc	Div
Round	Ln^	ErfInv	
Trunc	Log10^	ErfcInv (New)	
Rint	Log1p	CdfNorm (New)	
NearbyInt		CdfNormInv (New)	
Modf			

Figura 48: Funciones matemáticas incluidas en VML para redondeo.

Todas las funciones están disponibles sobre datos del tipo Real y ^ indica soporte adicional para datos del tipo Complex (complejos). ^^ Indica soporte solo para Complex

8.4.7 LIBRERÍA ESTADÍSTICA DE VECTORES

Incluye una gran cantidad de funciones estadísticas que se incluyen dentro de una librería denominada VSL (*Vector Statistical Library*).

- Convolución / Correlación: VSL contiene rutinas para realizar transformaciones de convoluciones y correlaciones lineales para precisión simple y doble.

- Generación de números aleatorios: Generación de números aleatorios básicos (BRNGs), Pseudo aleatorios, MCG59 (generador multiplicativo de 59 bit), MCG31m1 de 31 bit, MRG32k3a (generador recursivo múltiple de 32 bit), R250, Wichman-Hill, MT19937 (Marsenne). Adicionalmente también da soporte a generadores continuos (uniforme, Gaussian, exponencial, Laplace, Weibull, Cauchy, Rayleigh, Lognormal, Gumbel, Gamma, Beta, etc.) y con distribución discreta (uniforme, Bernoulli, geométrica, binomial, Poisson, etc.)

8.5 OPTIMIZACIÓN DE BUCLES

Los bucles suelen ser las zonas donde se consume más tiempo de cálculo en un código. Es por ello que entender perfectamente cómo se debe optimizar un bucle es fundamental para obtener el mejor rendimiento posible.

Para ello se debe conocer previamente dos conceptos fundamentales:

- Los diferentes métodos de indexación que se pueden realizar, para de esta forma analizar métodos de optimización de bucles basados en la indexación.

- Los principios de cómo funciona el *pipeline* en el procesador.

8.5.1 INDEXACIÓN

El índice de un bucle es el elemento clave dentro de un bucle.

Los índices que dan acceso a los vectores o matrices en los bucles deben expresarse de la forma más directa y explícita posible. Cuanto más fácil se deje al compilador, mejor podrá optimizar y con ello podrá generar código más óptimo.

A continuación se enumeran cuatro formas de acceso:

8.5.1.1 DIRECTO

El método directo de acceso consiste en expresar los índices de las matrices con los mismos índices que utilizamos para acceder a los bucles. En un apartado posterior se verá también que dependiendo del lenguaje que estemos utilizando (FORTRAN o C) resulta muy conveniente poner los índices en un orden adecuado para que se tenga mejor acceso a la caché del procesador.

Esta es la forma más óptima de escribir los accesos a matrices.

Un ejemplo de método directo en un bucle con acceso a una matriz es el siguiente:

```
Do j= 1, M
    Do i= 1, N
        a (i, j) = ...
    End Do
End Do
```

Como se puede comprobar, la matriz "a" se accede a través de los índices i y j que son los mismos que los utilizados en los dos bucles.

8.5.1.2 EXPLICITAMENTE CALCULADO

El método explícitamente calculado consiste en expresar los índices de las matrices mediante operaciones explícitas sobre los mismos índices que se utiliza para acceder a los bucles. Esta forma de acceso no es tan óptima como la anterior pero permite al compilador emplear técnicas de optimización para mejorar el rendimiento (como software *pipeline, unroll*, etc.).

Un ejemplo de acceso explícitamente calculado en un bucle con acceso a matriz es el siguiente:

```
Do j= 1, M
    Do i= 1, N
        a(( i-1)* N+ i) = ...
    End Do
End Do
```

Como se puede comprobar en el ejemplo, se está accediendo a la matriz "a" mediante operaciones directas sobre los índices o los límites de los índices. El compilador puede resolver en tiempo de compilación estas operaciones para optimizar los bucles.

8.5.1.3 ACUMULADO (INDUCCIÓN)

El método acumulado también conocido como inducción, consiste en emplear un índice diferente de los índices que se utilizan para acceder a los bucles.

Esta forma de acceso deja de ser óptima ya que no posibilita al compilador a utilizar técnicas para hacer un buen uso a priori de acceso a la caché.

Un ejemplo de acceso acumulado en un bucle con acceso a matriz es el siguiente:

```
k= 0

Do j= 1, M

    Do i= 1, N

        k= k+ 1

        a( k) = ...

    End Do

End Do
```

Como se puede comprobar en el ejemplo, se está accediendo en este caso a un vector mediante un índice diferente al utilizado en los dos bucles. En este caso el compilador tendrá dificultades para tratar de optimizar el código empleando software *pipeline* o técnicas de optimización de acceso a la caché.

8.5.1.4 **INDIRECTO**

El método indirecto es el peor método que se puede emplear para acceder a una matriz o vector en un bucle. Consiste en emplear como índice de acceso a la matriz un valor independiente de los índices de los bucles y que puede cambiar en tiempo de ejecución (como puede ser por ejemplo acceder mediante valores de otra matriz o vector).

Esta forma de acceso es la peor para un compilador puesto que le imposibilita generar código óptimo.

Un ejemplo de acceso indirecto es el siguiente:

```
Do j= 1, M

    Do i= 1, N

        k= k+ 1

        a( idx1( j)+ idx2( i)) = ...

    End Do

End Do
```

Como se puede comprobar en este ejemplo se está accediendo a la matriz a mediante la suma de los valores contenidos en 2 vectores idx1 e idx2. El compilador no puede saber qué valores contendrán los vectores idx1 e idx2 con lo cual no podrá en tiempo de compilación generar un código óptimo. Además con esta forma de escribir los accesos a las matrices se imposibilita seguramente un buen uso de la caché.

8.5.2 PIPELINE

Muchas de las técnicas que se van a ver son para mejorar el software *pipeline*. Es por ello que antes de analizar dichas técnicas se va a profundizar en este concepto.

Los bucles son implícitamente paralelos. Esto conlleva desde un paralelismo a nivel de instrucción hasta un paralelismo a nivel de datos para poder realizar las iteraciones del bucle sobre diferentes procesadores.

Consideremos el bucle siguiente el cual implementa una operación DAXPY:

```
do i = 1, n
        y(i) = y(i) + a*x(i)
enddo
```

Esta operación es la suma de un vector multiplicado por un escalar a un segundo vector, sobrescribiendo el segundo vector. Cada iteración de este bucle requiere las siguientes instrucciones:

- 2 LOAD: x(i) e y(i) y 1 store: y(i).
- Una suma-multiplicación.
- 2 incrementos de dirección.
- 1 test de final de bucle.
- 1 salto.

En una arquitectura escalar (no vectorial) el procesador puede ejecutar hasta un número X de instrucciones por ciclo de reloj. Pongamos que pueda ejecutar 4 instrucciones por ciclo de reloj con las siguientes combinaciones:

- 1 operación de load o store.
- 1 instrucción en la ALU 1 y 1 en la ALU 2.
- Una suma flotante.

- Una multiplicación.

Estas operaciones pueden ser ejecutadas en *pipeline*. El compilador puede utilizar técnicas para optimizar el rellenado de las diferentes unidades sin pérdidas de etapas. Cada iteración del bucle es rota en instrucciones que se sobrescriben para tener todas las unidades ocupadas en cada etapa, aunque esta operación no es siempre posible porque por ejemplo, no se tengan operaciones aritméticas sobre flotantes, o porque estén esperando un dato previo para operar, o simplemente porque sólo pueden ejecutar un número de instrucciones por ciclo. En estas ocasiones el compilador tiene que generar código con instrucciones vacías para rellenar el *pipeline* sobre las primeras y segundas iteraciones y sobre las últimas, así como código especial para aquellos bucles que requieran solamente de pocas iteraciones.

El *pipeline* es preparado para ser ejecutado muchas veces y así obtener mayor ventaja del mismo. El compilador puede determinar el mínimo número de ciclos que se requiere para obtener dicha ventaja en la parte principal del bucle e intentar programar las instrucciones para que al aumentar el número de bucles se aproveche al máximo el rendimiento.

Para calcular el mínimo número de ciclos requerido para el bucle, el compilador asume que todos los datos necesitados para una buena ejecución del *pipeline* debieran estar en caché (especialmente en caché primaria aunque las técnicas, tamaños, tipos, etc. de las cachés difieren en todos los procesadores del mercado). Si los datos no están en caché el bucle tomará mayor tiempo en ser ejecutado en cada iteración y no habrá aprovechamiento en la técnica de *pipeline*. Para evitar esto el compilador puede realizar otra técnica denominada *prefetch* o precarga.

La precarga permite al compilador adelantarse a la ejecución de operaciones aritméticas pidiendo a la memoria los datos por anticipado para que estén disponibles cuando se vaya a operar con ellos. Esta técnica permitirá disponer de los datos en el momento preciso con lo cual el *pipeline* no se retrasará y podrá ser ejecutado eficientemente.

En general considerar que los datos van a estar siempre en caché es una asunción poco realista; no obstante, en general, es sencillo determinar si el software *pipeline* ha sido ejecutado en el máximo rendimiento posible.

A continuación se va a mostrar un ejemplo muy típico de lo anteriormente expuesto mediante la operación DAXPI:

```
do i = 1, n
        y(i) = y(i) + a*x(i)
enddo
```

Supongamos que se dispone de un procesador que es capaz de ejecutar al menos una instrucción de memoria y una de suma-multiplicación por ciclo de reloj. Como se tiene

tres operaciones de acceso a memoria por cada operación de multiplicación-suma, el rendimiento óptimo de este bucle estaría en una tercera parte del pico flotante del procesador.

Para crear la sucesión de código, el compilador tiene que considerar muchas restricciones de hardware. Las siguientes condiciones son una muestra de las restricciones hardware que pueden afectar al bucle DAXPY:

- El número de *LOAD o STORE* que pueden ser ejecutadas por cada ciclo de reloj. Supongamos que sólo puede ser una.

- La latencia en la caché al cargar un dato flotante. Supongamos que es por ejemplo tres ciclos.

- Una operación *madd* realiza una multiplicación seguida de una suma y la carga del sumando puede empezar justamente un ciclo antes de la *madd*, ya que la multiplicación gastará 2 ciclos antes que el sumando sea necesitado.

- A los operandos de una instrucción de store se accede dos ciclos después por lo que el almacenamiento del resultado de una instrucción *madd* puede ser comenzado en 5-2 = 3 ciclos después que la operación *madd* comience.

Basándonos en estas mínimas consideraciones aunque podrían entrar en juego otras, el mejor esquema para realizar una iteración básica del bucle DAXPI sería la siguiente:

CICLO	INSTR1	INSTR2	INSTR3	INSTR4
0	Ld * x	++x		
1	Ld * y			
2				
3				Madd
4				
5				
6	St * y	Br	++y	

En este esquema, el bucle está ejecutando dos operaciones de coma flotante (multiplicación y suma en una instrucción) por cada 7 ciclos de reloj, es decir una séptima parte de pico potencial.

Este esquema pude ser mejorado realizando un *unroll* del bucle para que de esta forma calcule 2 vectores de elementos por iteración como se muestra a continuación:

```
do i = 1, n-1, 2

    y(i+0) = y(i+0) + a*x(i+0)

    y(i+1) = y(i+1) + a*x(i+1)
```

enddo

do i = i, n

y(i+0) = y(i+0) + a*x(i+0)

enddo

CICLO	INSTR1	INSTR2	INSTR3	INSTR4
0	Ld * (x + 0)			
1	Ld * (x + 1)			
2	Ld * (y + 0)	X+=2		
3	Ld * (y + 1)			Madd0
4				Madd1
5				
6	St * (y+0)			
7	St * (y+1)	Y+=2	br	

Este bucle ejecuta cuatro operaciones flotantes por ocho ciclos de reloj, o mejor dicho una cuarta parte del pico. Se comprueba que es más óptimo y que está cerca del máximo posible (un tercio) pero aún no es perfecto.

La forma más optima sería realizar un unroll del bucle de factor cuatro con lo cual se puede llegar al máximo posible ejecutando ocho operaciones flotantes por cada doce instrucciones, o una tercera parte del pico, consiguiendo de esta forma un ratio de 8/12 o mejor dicho una tercera parte del pico posible que era el rendimiento más óptimo posible.

CICLO	INSTR1	INSTR2	INSTR3	INSTR4
0	Ld y4			
1	St y0			
2	St y1			
3	St y2			

4	St y3			
5	Ld x5			
6	Ld y5			
7	Ld x6			
8	Ld x7			Madd4
9	Ld y6			Madd5
10	Ld y7	Y+=4		Madd6
11	Ld x0		Bne out	Madd7
0	Ld y0			
1	St y4			
2	St y5			
3	St y6			
4	St y7			
5	Ld x1			
6	Ld y1			
7	Ld x2			
8	Ld x3			Madd0
9	Ld y2			Madd1
10	Ld y3	Y+=4		Madd2
11	Ldx4		Beq loop	Madd3

Como se puede ver este esquema, consiste de dos bloques de doce ciclos. Cada bloque de doce ciclos se llama una replicación puesto que el mismo código es repetido en cada bloque. Cada replicación permite acoplar cuatro iteraciones del bucle, conteniendo cuatro operaciones madd y doce operaciones de memoria, así como un incremento de puntero y un salto, permitiendo de esta forma poder ejecutar el máximo pico nominal para este bucle.

INICIO	REPLICACIÓN 0	REPLICACIÓN 1,2,..N	FINAL

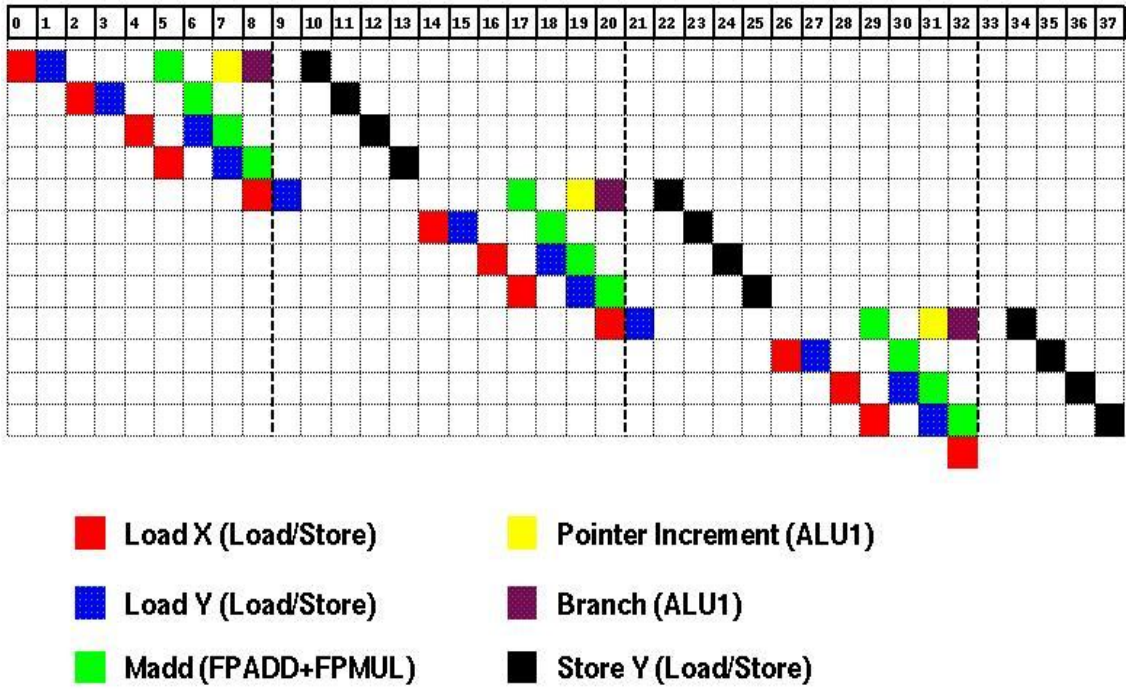

Figura 49: Ejemplo de *pipeline.*

Las primeras cinco iteraciones llenan el software *pipeline*. Este es el estado de inicio; se omitió del ejemplo anterior por simplicidad. Los ciclos del nueve hasta el veinte realizan la primera replicación. Ciclos del veintiuno hasta el treinta y dos, la segunda replicación, y así hasta el final.

El compilador ha desenvuelto el bucle cuatro veces. Cada iteración es extendida sobre varios ciclos: quince para la primera iteración de las cuatro iteraciones desenvueltas, diez para el segundo y nueve para las iteraciones tres y cuatro. Como solamente una operación a memoria puede ser realizada por ciclo, las cuatro iteraciones deben ser compensadas desde unas a otras, tomando dieciocho ciclos para completar las cuatro. Se necesitan dos replicaciones para contener los dieciocho ciclos completos de trabajo para las cuatro iteraciones.

Un test al final de cada iteración determina si todos los posibles grupos de cuatro iteraciones han sido comenzados. Si no, la ejecución los lanza a la próxima replicación o salta de vuelta a la primera replicación al comienzo del bucle. Si todos los grupos de cuatro iteraciones han comenzado, la ejecución salta al código que finaliza a la iteración en progreso. Esto es el final del código que por simplicidad también fue omitido en el ejemplo anterior.

Para ayudar a determinar cómo de efectivo el compilador ha sido capaz de realizar el software *pipeline*, los compiladores son capaces en su mayoría de generar un reporte de cada bucle que ha optimizado. Mediante una opción en el compilador (en muchos casos esta opción es –S) se puede generar un listado entendible e incluso generar la salida en ensamblador puro (lo cual en muchos casos requiere de grandes conocimientos de ensamblador para tratar de comprender cómo el compilador ha generado el código).

En el ejemplo que estamos tratando se ha generado un reporte para un compilador en concreto (MipsPro Compiler de Silicon Graphics):

#<swps>

#<swps> Pipelined loop line 6 steady state

#<swps>

#<swps> 50 estimated iterations before pipelining

#<swps> 2 unrollings before pipelining

#<swps> 6 cycles per 2 iterations

#<swps> 4 flops (33% of peak) (madds count as 2)

#<swps> 2 flops (16% of peak) (madds count as 1)

#<swps> 2 madds (33% of peak)

#<swps> 6 mem refs (100% of peak)

#<swps> 3 integer ops (25% of peak)

#<swps> 11 instructions (45% of peak)

#<swps> 2 short trip threshold

#<swps> 7 ireg registers used.

#<swps> 6 fgr registers used.

Este listado no muestra como de eficiente el compilador ha utilizado los recursos hardware para generar el software *pipeline* para este bucle. Las líneas muestran cuantas instrucciones de cada tipo hay por cada replicación y que porcentaje de utilización del pico han sido ejecutados.

El listado muestra que empezando el bucle en la línea seis del código fuente (esta línea puede ser aproximada) el bucle ha sido desenvuelto dos veces. El *scheduler* generado

por el compilador requiere de seis ciclos de reloj por replicación y cada replicación completa dos iteraciones a causa del *unrolling*.

Como se puede apreciar, el rendimiento de este bucle es del 33% del pico en coma flotante. La línea "*madds count as 2*" nos refleja la cuenta de operaciones en coma flotante. Esto indica cuan de rápido pude ejecutarse este bucle en una escala absoluta. El ratio de pico del procesador es dos operaciones flotantes para el procesador MIPS con lo cual podrían generar en un procesador a 800 Mhz (el último procesador MIPS) 1600 Mflops de pico. Por lo tanto se está ejecutando 533 Mflops.

8.5.3 TÉCNICAS DE OPTIMIZACIÓN DE BUCLES

Basándonos en los criterios anteriormente mencionados a continuación se describen las técnicas más comunes de optimización de bucles:

- Intercambio del orden de los bucles (*loop interchange*).
- Desenrollar bucles (*loop unrolling*).
- Bloques en bucles (*loop blocking*).
- Fusión de bucles (*loop fusion*).
- División de bucles (*loop splitting*).
- Intercambio de IF-DO.
- Recortar bucles (*loop peeling*).
- Inducción de variables en los bucles.
- Romper dependencia de datos en los bucles.

8.5.4 INTERCAMBIO DEL ORDEN DE BUCLES ANIDADOS

8.5.4.1 DESCRIPCIÓN

La técnica de intercambio del orden de bucles anidados consiste en comprobar si se ha programado el código de tal forma que los elementos de una matriz sean accedidos de la misma forma en que han sido almacenados en memoria.

El orden en el cual el ordenador almacena los datos en la memoria depende del lenguaje de programación utilizado. Mayoritariamente los códigos científicos están programados en lenguaje Fortran o en C/C++.

En Fortran, los datos en una matriz se almacena por columnas:

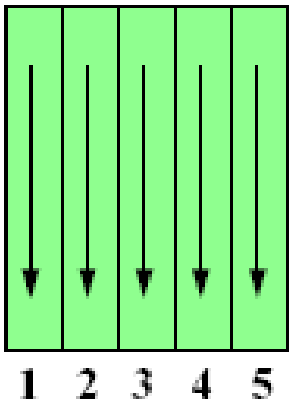

Figura 50: Almacenamiento en Fortran.

Sin embargo en C/C++ los datos de una matriz se almacenan por filas:

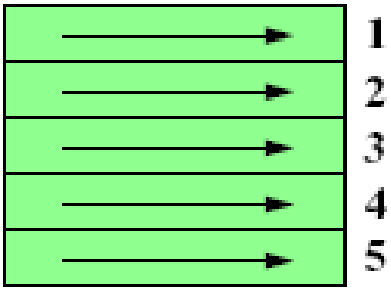

Figura 51: Almacenamiento en C/C++.

Por lo tanto, cuando se plasma un algoritmo de pseudocódigo a código de programación (bien sea Fortran, C u otro tipo de programación) hay que tener en cuenta la forma que este lenguaje está almacenando los datos en memoria para que al ser accedidos se haga en el mismo orden y de esta forma tengamos un buen uso y por tanto una buena reutilización de los datos en caché, fundamental para acelerar el acceso a memoria y con ello la rapidez de un código.

8.5.4.2 **VENTAJAS**

Las ventajas de esta técnica son:

- Tener un mejor patrón de acceso a la memoria debido a una mejor utilización de las líneas de caché.

- Eliminación de dependencias de datos y con ello un incremento de las oportunidades de optimización y paralelización.

- Mejorar la localidad de referencia bien sea espacial que es la tendencia de un proceso a referenciar una porción del espacio virtual de direcciones cercano a la última referencia o bien la temporal que es la tendencia de un proceso a referenciar en el futuro elementos referenciados en un pasado reciente.

8.5.4.3 **DESVENTAJAS**

Las desventajas de esta técnica son:

- Puede dejar el bucle más corto en el interior, lo cual no es lo más adecuado para el software *pipeline.*

8.5.4.4 **EJEMPLO 1**

A continuación se muestra un bucle muy típico donde se suman dos matrices y se almacenan de nuevo en la primera matriz:

```
Do i= 1, N
    Do j= 1, M
        A( i, j ) = A( i, j ) + B( i , j )
    End Do
End Do
```

En Fortran el orden de almacenamiento es por columnas por lo que estos bucles se deben organizar para que el acceso a las matrices se haga de la misma forma que se han almacenado.

Si se expresa en Fortran el bucle como se ha escrito, se está accediendo al revés de cómo se ha almacenado:

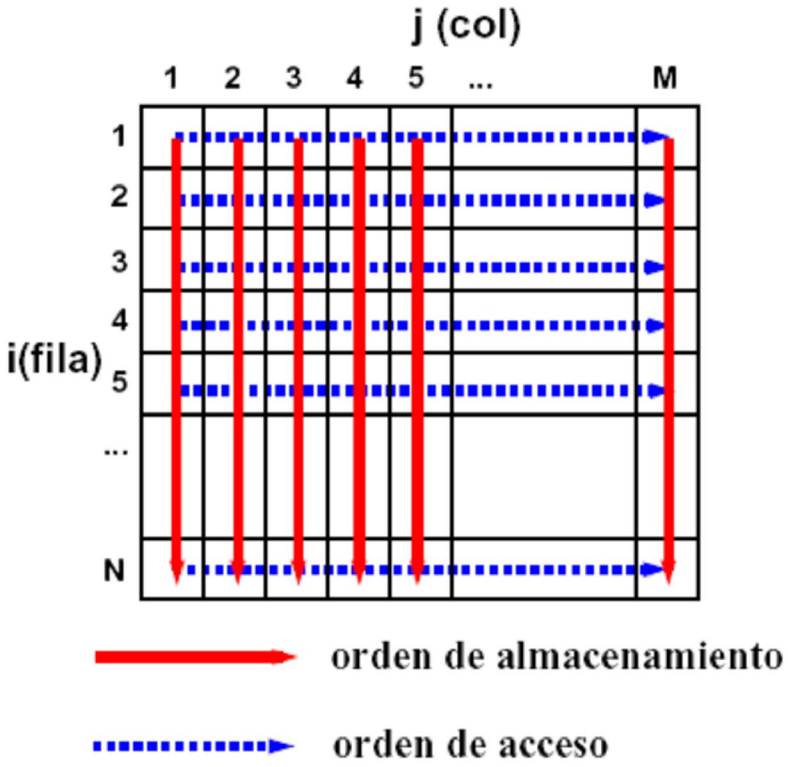

Figura 52: Orden de acceso incorrecto.

Es por ello que se debe dar la vuelta a los bucles de la siguiente forma:

```
Do j= 1, M
        Do i= 1, N
                A( i, j ) = A( i, j ) + B( i, j )
        End Do
End Do
```

O bien dar la vuelta a los índices dentro de la matriz:

```
Do i= 1, N
        Do j= 1, M
                A( j, i ) = A( j, i ) + B( j , i )
        End Do
End Do
```

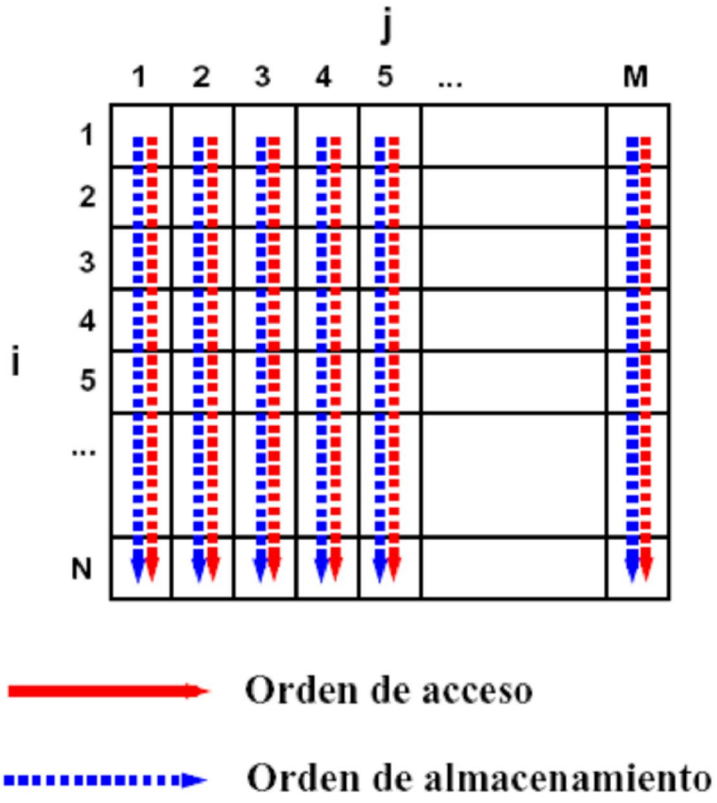

Figura 53: Orden de acceso correcto.

Así se consigue localidad espacial de referencia.

Sin embargo, si se programa en C, el orden de almacenamiento es al revés que en Fortran, ya que se almacena por filas.

Ejemplo de mala localidad de referencia en C:

```
for (j= 0; j< M; j++)
        for (i= 0; i< N; i++)
                C[ i][ j] = A[ i][ j] + B[ i][ j];
```

En C se debería o bien cambiar el índice en las matrices:

```
for (j= 0; j< M; j++)
        for (i= 0; i< N; i++)
                C [j][ i] = A[ j][ i] + B[ j][ i];
```

O bien cambiar el orden a los bucles:

```
for (i= 0; i< N; i++)
        for (j= 0; j< M; j++)
                C[ i][ j] = A[ i][ j] + B[ i][ j];
```

Como resumen, hay que decir que en Fortran se debe programar los bucles con los índices cruzados mientras que en C se debe programar los bucles con los índices sin cruzar.

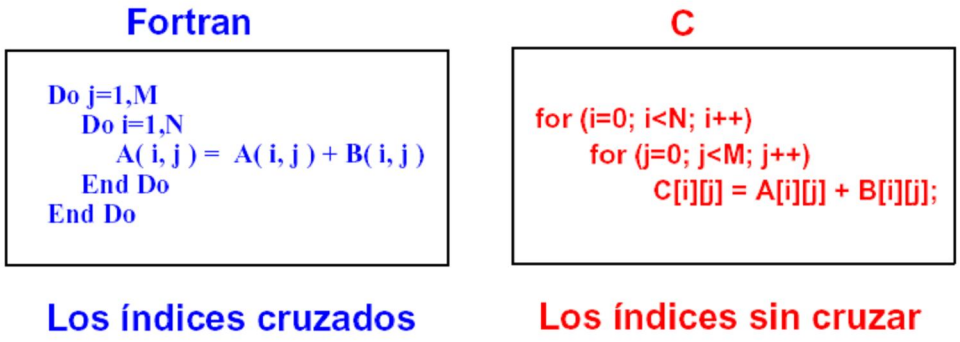

Figura 54: Resumen de cómo programar bucles en C y Fortran.

En general el intercambio de bucles o el intercambio de índices tienen el mismo beneficio en rendimiento, pero en muchas situaciones prácticas el intercambio de bucles es más fácil de implementar.

8.5.4.5 EJEMPLO 2

```
COMMON /SHARE/ B (M, N)

.....

SUBROUTINE DO_ EJEMPLO2( N, M, A)

REAL A( N, M)

DO j= 1, M

        DO i= 1, N

                A( i, j) = A( i, j) + B( j, i)

        END DO

END DO
```

El intercambio de bucles sería bueno para la matriz B pero malo para la matriz A.

Para ello lo que hacemos es dar la vuelta al orden de almacenamiento de la matriz B:

```
COMMON /SHARE/ B (N, M)
.....
SUBROUTINE DO_ EJEMPLO2 ( N, M, A)
REAL A( N, M)
DO j= 1, M
        DO i= 1, N
                A( i, j) = A( i, j) + B( i, j)
        END DO
END DO
```

Lo que se ha hecho es dar la vuelta al índice en B reemplazando B(j, i) por B(i, j).

8.5.4.6 **EJEMPLO 3**

Los 2 ejemplos anteriores no consideraban la posibilidad que hubiera iteraciones que necesitaran de un ajuste especial. En este ejemplo se analiza este caso:

```
DIMENSION A( N, N, N), B( N, N, N), C( N, N)
DO i= 1, N
        DO j= 1, N
                DO k= 1, N
                        A( i, j, k) = A( i, j, k) + B( i, j, k) * C( i, j)
                END DO
        END DO
        C( N, i) = C( i, N) + Cst1
END DO
```

En este caso:

- A, B y C son accedidos en la dimensión incorrecta.
- Por tanto interesa hacer un intercambio de bucles.
- Sin embargo existe una relación delicada entre A y C.

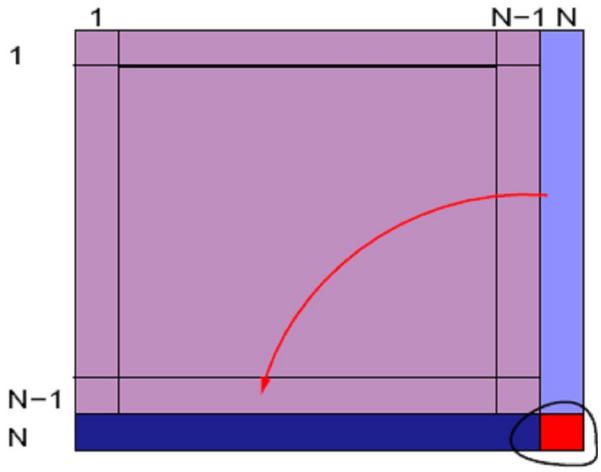

Figura 55: Ejemplo de estudio particular de un elemento en una matriz.

La última fila de C es modificada, por lo que se tiene que dividir las operaciones que usan C(N, 1... N.1) y C(N, N), tendiendo en cuenta que el último elemento necesita de una especial atención:

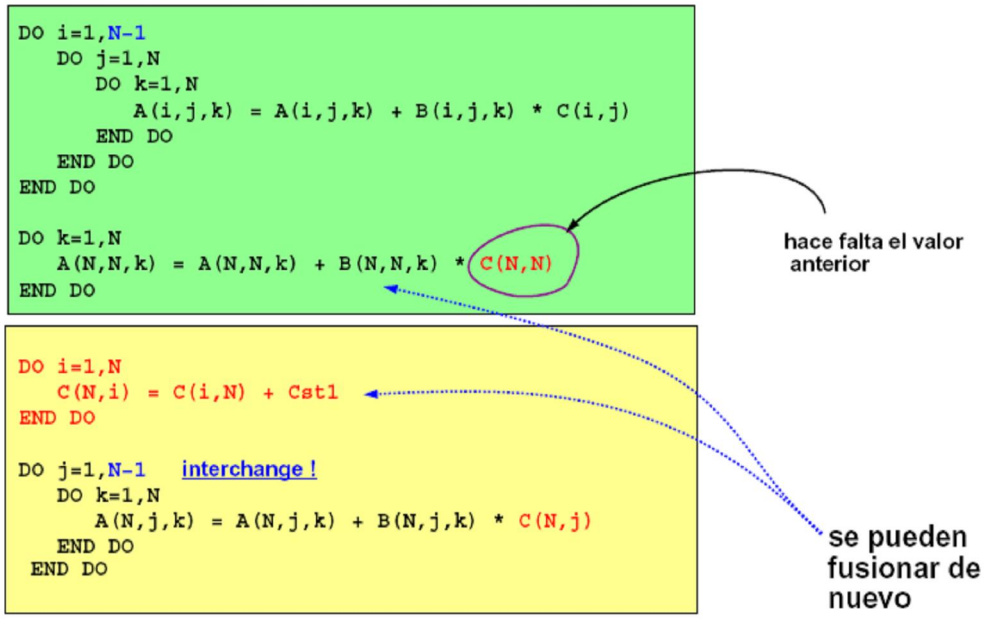

Figura 56: Ejemplo de fusión e intercambio.

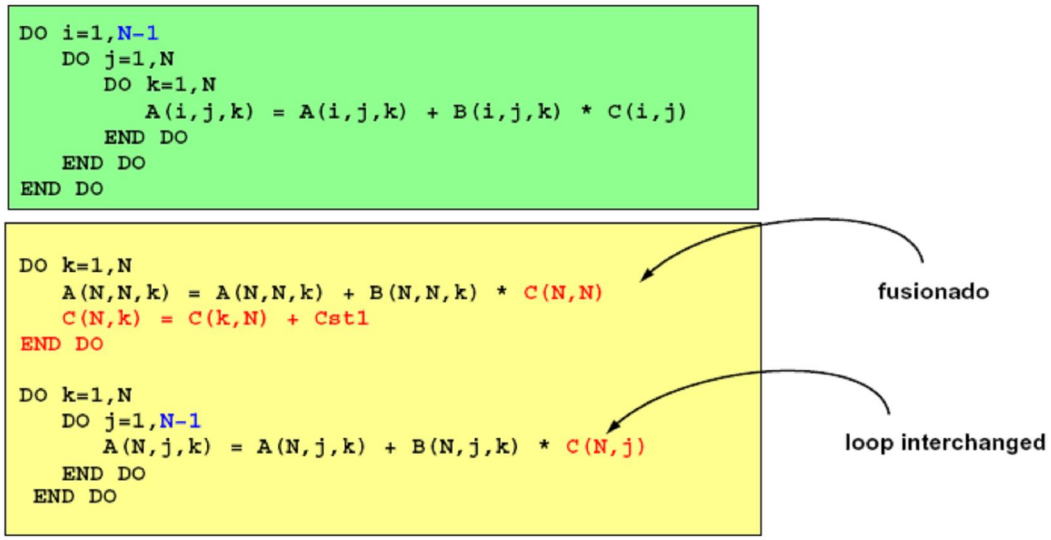

Figura 57: Resultado final de fusión e intercambio en un bucle.

Con este ejemplo, se puede llegar a conseguir un *speedup* del 20x con respecto al código original.

8.5.5 DESENROLLAR BUCLES

8.5.5.1 DESCRIPCIÓN

La técnica de desenrollar bucles (*loop unrolling*) consiste en implementar directamente dentro de un bucle secuencial N iteraciones de una vez y que el bucle se realice de N iteraciones en N iteraciones en vez de 1 en 1.

DO i = 1, N, 1

...(i)...

END DO

Se transformaría en:

DO i = 1, N, unroll

...(i)...

...(i+1)...

```
...(i+2)...

...(i+unroll−1)...
```

END DO

8.5.5.2 **VENTAJAS**

Las ventajas de esta técnica son:

- Más oportunidades para elaborar un código superescalar.
- Mejor reutilización de datos.
- Explotar los datos en las líneas de caché.
- Reducción en la sobrecarga del bucle. El incremento, comparación y ramificación de los índices del bucle suelen llevar el mayor tiempo de ejecución por iteración.

8.5.5.3 **DESVENTAJAS**

Las desventajas de esta técnica son:

- Se necesitan más registros del procesador
- Se necesita hacer una corrección en las iteraciones finales para garantizar el número correcto de iteraciones:

  do i= 1, N mod(N, unroll)), unroll & do i= N mod(N, unroll)+ 1,N

8.5.5.4 **EJEMPLO**

```
DO I = 1, N
    DO J = 1, N
        A( I) = A( I) + B( I, J)* C( J)
    END DO
END DO
```

Desenrollamos el bucle en un factor 4:

```
DO I = 1, N, 4
    DO J = 1, N
        A( I) = A( I) + B( I ,J) * C( J)
        A( I+ 1) = A( I+ 1) + B( I+ 1, J) * C( J)
        A( I+ 2) = A( I+ 2) + B( I+ 2, J) * C( J)
        A( I+ 3) = A( I+ 3) + B( I+ 3, J) * C( J)
    END DO
END DO i
```

Es importante destacar que el factor debe cuadrar con el tamaño de la línea de caché para que además un intercambio de bucles dé adicionalmente una mejor referencia a memoria.

8.5.6 BLOQUES DE BUCLES

8.5.6.1 DESCRIPCIÓN

La técnica de implementar los bucles en bloques (*loop blocking*) consiste en dividir las matrices y vectores en bloques debido a que toda la matriz no cabe en caché.

```
DO i = 1, N
...i...
```

Pasaría a:

```
DO i1 = 1, N, Nblocking
    DO i2 = 0, min(N−i1+1, Nblocking)−1
        ...i = i1 + i2...
```

8.5.6.2 VENTAJAS

- Se incrementa la localidad de referencia temporal (reutilización de los datos).

- Se incrementa la localidad espacial de referencia (explotar las líneas de caché completamente).

8.5.6.3 DESVENTAJAS

Las desventajas de esta técnica son:

- Puede resultar complejo o incluso imposible permutar las dimensiones de los *arrays* o intercambiar bucles.
- Puede existir problemas como a[i][j] = b[j][i] que nos impide realizarlo.
- Genera un código oscuro de leer.

8.5.6.4 EJEMPLO 1

```
for (j=0; j<N; j++)
        for (i=0; i<N; i++)
                A[i][j] += B[i][j
```

Orden de almacenamiento en línea de caché →

Orden de acceso ↓

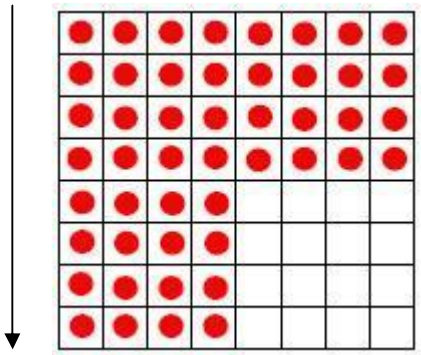

Figura 58: Ejemplo de descomposición de bloques en una matriz.

Como se puede apreciar, todos los elementos están accediendo fuera de caché. Son aquellos que aparecen en color rojo.

Ahora se diseña en bloques para que la mayoría de elementos (en azul) estén dentro de caché, mientras que una minoría (los de color rojo) están fuera de caché:

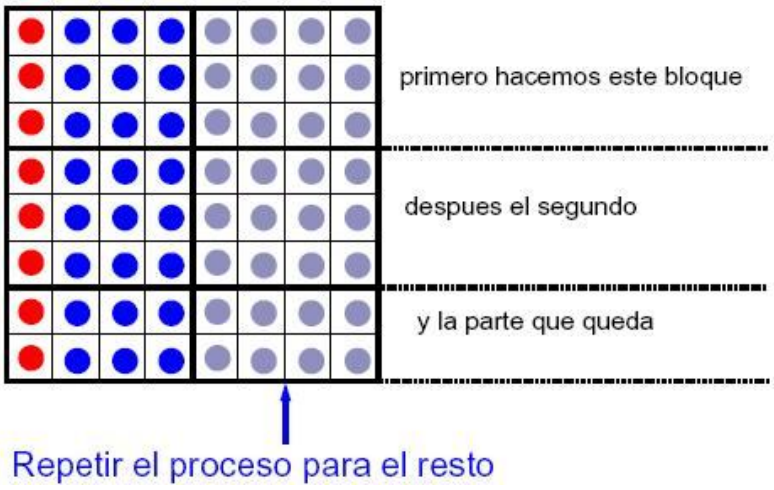

Figura 59: Resumen de descomposición de bloque para mejor acceso a caché.

8.5.6.5 EJEMPLO 2: MULTIPLICACIÓN DE MATRICES

Esta técnica es muy utilizada en algoritmos de multiplicación de matrices (C= A * B)

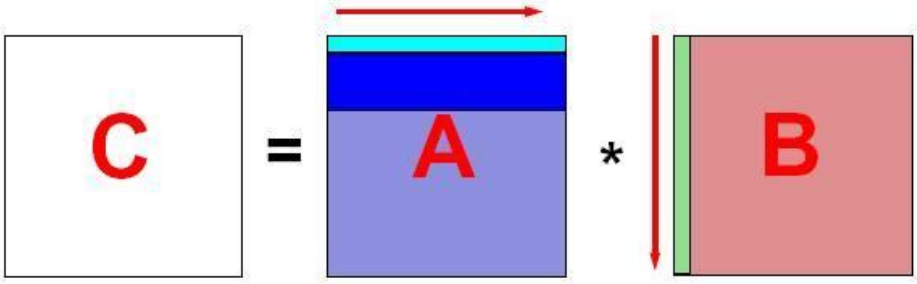

Figura 60: Multiplicación de matrices.

Para tamaños grandes de matrices A, B y C, la caché no suele ser suficiente y se produce la llamada contención de caché.

Por ello, lo que se hace es programar el código en bloques que quepan dentro de la caché:

Figura 61: Descomposición en bloques de multiplicación de matrices.

C11 = A11 * B11 + A12 * B21

C12 = A11 * B12 + A12 * B22

Con el tamaño de bloque apropiado, A11 y A12 pueden caber dentro de caché y se pueden reutilizar para calcular C.

Como línea general, **cuando el número de bucles es mayor el número de dimensiones** existe la posibilidad de realizar esta técnica.

8.5.7 FUSIÓN DE BUCLES

8.5.7.1 DESCRIPCIÓN

La técnica de fusión de bucles (*loop fusion*) consiste en unir dos o más bucles en uno solo.

```
DO
    Trabajo 1
END DO
DO
    Trabajo 1
END DO
```

Se junta en un solo bucle:

```
DO
    Trabajo 1
    Trabajo 2
END DO
```

8.5.7.2 VENTAJAS

Las ventajas de esta técnica son:

- Reutilización de los datos de la caché.

- Menor sobrecarga de bucles.
- Mayor simplicidad.
- Mejor paralelización debido a menor sobrecarga de creación de *threads* en OpenMP.

8.5.7.3 **DESVENTAJAS**

- Aumento de los datos que compiten por la caché.
- Mayor número de registros.

8.5.7.4 **EJEMPLO**

```
DO i = 1, N
        B (i) = 2 * A (i)
END DO
DO k = 1, N
        C (k) = B (k) + D (k)
END DO
```

Como los índices de los dos bucles coinciden en el tamaño (de 1 a N) se pueden juntar los 2 bucles en uno solo:

```
DO ii = 1, N
        B( ii) = 2 * A( ii)
        C( ii) = B( ii) + D( ii)
END DO
```

Con ello, se está reutilizando el vector B.

8.5.8 DIVISIÓN DE BUCLES

8.5.8.1 DESCRIPCIÓN

La técnica de división de bucles (*loop splitting*) consiste en dividir un bucle en dos o más bucles. Es el caso contrario al anterior visto.

```
DO
    Trabajo 1
    Trabajo 2
END DO
```

Se divide en dos bucles:

```
DO
    Trabajo 1
END DO
DO
    Trabajo 2
END DO
```

8.5.8.2 VENTAJAS

Las ventajas de esta técnica son:

- Se puede optimizar mejor por separado cada uno de los bucles aplicando otras técnicas.

8.5.8.3 DESVENTAJAS

- Se crean varios bucles con el consiguiente aumento de sobrecarga de los bucles.

8.5.8.4 EJEMPLO

```
DO i = 1, N
        B (i) = 2 * A (i)
        D (i) = D (i-1) + C (i)
END DO
```

Como los índices de los vectores coinciden con el tamaño de los índices (de 1 a N) se pueden dividir el bucle en dos 2 bucles:

```
DO ii = 1, N
        B( ii) = 2 * A( ii)
END DO
DO i = 1, N
        D(i) = D(i-1) + C(i)
END DO
```

Sobre el primer bucle se puede implementar otras técnicas de optimización como por ejemplo paralelización ya que cuando estaban todo en un mismo bucle se producía una dependencia de datos.

8.5.9 INTERCAMBIO DE IF-DO

8.5.9.1 INTRODUCCIÓN

Esta técnica consiste en sacar fuera del bucle aquellas sentencias del estilo IF, CASE, etc.

```
DO I = 1,N
        IF xxx THEN …
        ELSE
        ENDIF
ENDDO
```

Se transformaría en:

```
IF xxx THEN
        DO I = 1,N
        …
        ENDDO
ELSE
        DO I = 1,N
        …
        ENDDO
ENDIF
```

Los saltos pueden impactar negativamente en el rendimiento ya que interfieren con el *pipelining*.

8.5.9.2 VENTAJAS

Las ventajas de esta técnica son:

- Mejorar el *pipeline.*

8.5.9.3 DESVENTAJAS

- No hay desventajas en esta técnica.

8.5.9.4 EJEMPLO

```
DO I = 1,N
        IF (A. GT. 0) THEN
                X(I) = X(I) + 1.0
        ELSE
                X (I) = 0.0
```

```
        ENDIF
ENDDO
```

Si se intercambia el IF y el DO la sentencia IF se ejecuta solamente una vez:

```
IF (A. GT. 0.0) THEN
        DO I = 1,N
                X(I) = X(I) + 1.0
        ENDDO
ELSE
        DO I = 1,N
                X(I) = 0.0
        ENDDO
ENDIF
```

8.5.10 RECORTAR BUCLES

8.5.10.1 INTRODUCCIÓN

Esta técnica consiste en analizar elementos especiales dentro de un bucle (suele ser el primer o el último elemento del bucle) que causan que se programe de una forma especial introduciendo sentencias IF dentro del bucle u otra serie de programaciones deficientes.

8.5.10.2 VENTAJAS

- Se aumenta el rendimiento en el *pipeline.*
- Se puede llegar a paralelizar.

8.5.10.3 **DESVENTAJAS**

No hay desventajas en esta técnica.

8.5.10.4 **EJEMPLO1**

En el siguiente ejemplo dependiendo de los límites de condición del bucle, se está realizando unas operaciones u otras:

```
DO I = 1,N
   IF (I .EQ. 1) THEN
     X(I) = 0
   ELSEIF (I .EQ. N) THEN
     X(I) = N
   ELSE
     X(I) = X(I) + Y(I)
   ENDIF
ENDDO
```

Esto se puede cambiar cortando el bucle:

```
X(1) = 0
DO I = 2, N-1
  X(I) = X(I) + Y(I)
ENDDO
X(N) = N
```

8.5.10.5 **EJEMPLO 2**

El siguiente ejemplo es muy típico cuando se utiliza un vector para similar coordinadas cilíndricas donde el valor de la izquierda del vector debe ser adyacente al de la derecha.

Se suele utilizar para ello una variable "*wraparound*"

```
jn1 = n;

for ( j=0; j<n; j++ ) {

        b[j] = (a[j] + a[jn1]) / 2;

        jn1 = j;

}
```

En la primera iteración, jn1 es n. En todas las demás iteraciones excepto para j=0, el valor de jn1 es j-1.

Luego jn1 es una variable a inducir para el bucle después de la primera iteración.

Si cortamos la primera iteración del bucle, la variable jn1 puede ser inducida:

```
b[0] = (a[0] + a[n]) / 2;

for ( j = 1; j<n; j++ ) {

        b[j] = (a[j] + a[j-1]) / 2;

}
```

8.5.11 INDUCCIÓN DE VARIABLES EN EL BUCLE

8.5.11.1 INTRODUCCIÓN

La inducción de variables en el bucle consiste en la localización de variables que son incrementadas o decrementadas en la misma cantidad para cada iteración.

Una variable x es inducida cuando se modifica de la forma x=x+K, donde K es invariante en el bucle. Lógicamente la operación puede ser una suma, resta, multiplicación, etc.

Analizando su relación con las variables de control del bucle, permite romper dependencias y posibilitar optimizaciones.

8.5.11.2 VENTAJAS

- Aumentar el *pipeline.*

8.5.11.3 DESVENTAJAS

- No se han prescrito desventajas.

8.5.11.4 EJEMPLO

```
j=k=0
for ( i=0; i<n; i++ ) {
        a[j] = b[k];
        j=j+1
        k=k+2;
}
```

Se convierte en:

```
j = k = 0;
for ( i = 0; i<n; i++ ) {
        a[j+i] = b[k+i*2];
}
```

8.5.12 ROMPER LA DEPENDENCIA DE DATOS EN LOS BUCLES

8.5.12.1 INTRODUCCIÓN

Esta técnica consiste en localizar relaciones entre variables que no dependen necesariamente de los índices de los bucles.

8.5.12.2 VENTAJAS

- Aumentar el *pipeline.*

8.5.12.3 **DESVENTAJAS**

- No se han prescrito.

8.5.12.4 **EJEMPLO**

```
p = n + 1;

for ( i=0; i<m; i++ ) {

        a[i][n] = a[i-1][p];

}
```

Se transforma en:

```
for ( i = 0; i<m; i++ ) {

        a[i][n] = a[i-1][n+1];

}
```

Como se puede apreciar, la dependencia entre n y p puede ser rota para conseguir acelerar el *pipeline.*

8.5.13 **OTRAS GUÍAS GENERALES PARA OPTIMIZAR BUCLES**

Buenas prácticas generales para los bucles son las siguientes:

- Mantener el tamaño de los bucles manejable.
- Acceder a los datos secuencialmente. Siempre que sea posible, organizar los índices de los bucles de la misma manera que se está accediendo a memoria para poder reutilizar la caché.
- Sacar fuera de los bucles sentencias IF que no sirvan para nada.
- Evitar llamar a funciones o subrutinas dentro de un bucle. Esto evita el coste del salto y retorno. Usar el código de la rutina que se estaba llamando dentro del bucle o reemplazarlo por una subrutina que contenga el bucle.
- Simplificar los *array subscripts*, especialmente expresiones que involucran a variables del bucle.

- Usar variables locales del tipo *INTEGER*.

- Evitar el uso de *I/O statements* en el bucle. Llamadas a funciones de E/S puede eliminar la optimizaciones de paralelismo a nivel de instrucción en procesadores *out-of-order*.

8.5.14 EJEMPLO GLOBAL: MULTIPLICACIÓN DE MATRICES

Como caso práctico y de estudio, se va a analizar una de las operaciones más típicas en códigos científicos como es la multiplicación de matrices.

```
DO i= 1, m
    DO j= 1, n
        DO k= 1, p
            C( i, j)= C( i, j)+ A( i, k)* B( k, j)
        END DO
    END DO
END DO
```

8.5.14.1 PASO 1. DIAGNÓSTICO

Los primeros problemas que nos encontramos son:

- C(i, j) es invariante y se puede sacar del bucle.

- El acceso a memoria para la matriz A es malo, pero sin embargo para B es bueno.

8.5.14.2 PASO 2. PRIMERA OPTIMIZACIÓN CAMBIO A CONSTANTE E INTERCAMBIO DE BUCLES

Esta técnica todavía no se ha visto y se verá posteriormente en las técnicas misceláneas.

Básicamente consiste en sacar de un bucle aquellas operaciones que son constantes y se puede realizar sólo una vez en vez de realizarla en las N iteraciones.

Por otro lado, se intercambia los bucles i, j, k a j, k, i para mejor acceso de los elementos A, B y C. Ahora los elementos de la matriz A, son accedidos en el orden correcto en Fortran, al revés de cómo fueron almacenados.

Lo mismo ocurre con los elementos de las matrices B y C.

```
DO j= 1, n
        DO k= 1, p
        t = B( k, j)
                DO i= 1, m
                        C( i, j)= C( i, j)+ A( i, k)* t
                END DO
        END DO
END DO
```

8.5.14.3 PASO 3. SEGUNDA OPTIMIZACIÓN: DESENROLLAR EL BUCLE INTERNO

```
DO j= 1, n
        DO k= 1, p
        t= B( k, j)
                DO i= 1,4* nb, 4
                        C( i ,j)= C( i ,j)+ A( i ,k)* t
                        C( i+ 1, j)= C( i+ 1, j)+ A( i+ 1, k)* t
                        C( i+ 2, j)= C( i+ 2, j)+ A( i+ 2, k)* t
                        C( i+ 3, j)= C( i+ 3, j)+ A( i+ 3, k)* t
                END DO
                DO i= 4* nb+ 1, m
                        C( i, j)= C( i, j)+ A( i, k)* t
```

```
            END DO
        END DO
    END DO
```

Al desenrollar este bucle, se está reutilizando el elemento t en la caché.

8.5.14.4 PASO 4. TERCERA OPTIMIZACIÓN: DESENROLLAR EL BUCLE INTERMEDIO

Se desenrolla el bucle intermedio para tener menos *load/stores* en la matriz C(i, j).

```
DO j= 1, n
    DO k= 1,4* nb, 4
        t0= B( k ,j); t1= B( k+ 1, j)
        t2= B( k+ 2, j); t3= B( k+ 3, j)
        DO i= 1, m
            C( i, j)= C( i, j)+ A( i, k) * t0
            + A( i, k+ 1) * t1
            + A( i, k+ 2) * t2
            + A( i, k+ 3) * t3
        END DO
    END DO
    DO k= 4* nb+ 1, p
        t= B( k, j)
        DO i = 1, m
            C( i, j)= C( i, j)+ A( i, k)* t
        END DO
    END DO
END DO
```

8.5.14.5 PASO 4. CUARTA OPTIMIZACIÓN: DESENROLLAR EL BUCLE SUPERIOR

Al desenrollar el bucle superior, se está haciendo un reutilización de los elementos de A (i, k) y de t0, t1, t2 y t3.

```
DO j= 1,4* nb, 4
        DO k= 1, p
                t0= B( k, j); t1= B( k, j+ 1)
                t2= B( k, j+ 2); t3= B( k, j+ 3)
                DO i= 1, m
                        C( i, j )= C( i, j )+ A( i, k) * t0
                        C( i, j+ 1)= C( i, j+ 1)+ A( i, k) * t1
                        C( i, j+ 2)= C( i, j+ 2)+ A( i, k) * t2
                        C( i, j+ 3)= C( i, j+ 3)+ A( i, k) * t3
                END DO
        END DO
END DO
DO j= 4* nb+ 1, n
        DO k= 1, p
                t= B( k, j)
                DO i = 1, m
                        C( i, j)= C( i, j)+ A( i, k)* t
                END DO
        END DO
END DO
```

8.5.14.6 PASO 5. QUINTA OPTIMIZACIÓN: IMPLEMENTAR BLOQUES EN EL BUCLE

Al implementar la técnica de bloque, se está reutilizando A(i, k) por cada valor de j.

```
DO j1 = 1, n, Nbj
    DO k1 = 1, p, Nbk
        DO i1 = 1, m, Nbi
            DO k2 = 0, min( p.k1+ 1, Nbk) .1
                DO j2 = 0, min( n.j1+ 1, Nbj) .1
                    k = k1+ k2
                    j = j1+ j2
                    t = B( k, j)
                    DO i2 = 0, min( m.i1+ 1, Nbi) .1
                        i = i1 + i2
                        C( i, j)= C( i, j)+ A( i, k)* t
                    END DO
                END DO
            END DO
        END DO
    END DO
END DO
```

El tamaño de bloque óptimo es una función dependiente de la longitud del bucle y del número de *arrays* involucrados y su tipo. Para el ejemplo que se está analizando:

- Tenemos tres matrices de tamaño NBxNB.
- Cada elemento es de precisión doble y por tanto ocupa ocho bytes.
- La condición para NB: 3* NB* NB* 8 < Tamaño_Caché_ En_ Bytes.

- Por tanto: NB < SQRT(Tamaño_Caché_ En_ Bytes / 24)
- Se debe redondear por debajo de un 10% menos para almacenar variables escalares

8.5.14.7 PASO 6. TOMA DE TIEMPOS

Se ha tomado tiempos de todas las versiones de modificaciones:

1. código original.

2. loop reversal y loop constant.

3. loop reversal y loop unrolling (interior).

4. loop reversal y loop unrolling (medio).

5. loop reversal y loop unrolling (exterior).

6. loop reversal y loop unrolling (medio y exterior).

7. loop reversal y unrolling (medio y exterior) con blocking.

Sobre un Xeon a 2 Ghz sin opción de optimización y activando en otro caso el software *pipeline*, los resultados en segundos fueron los siguientes:

OPTIMIZACIÓN	SIN FLAG DE OPTIMIZACIÓN	ACTIVANDO SWP
Código Original	15,6	88,3
LR + Lcte	8,84	8,7
LR + LU (interior)	8,27	8,73
LR + LU (medio)	5,65	6,00
LR + LU (externo)	3,77	3,74
LR + LU (medio + exterior)	3,52	3,51
LR + LU (medio + exterior)+LB	1,53	1,51
-O3	1,50	

Figura 62: Tabla resumen de diferentes optimizaciones para multiplicación de matrices.

8.6 ANÁLISIS INTERPROCEDURAL

Se denomina análisis interprocedural a aquellas optimizaciones que se efectúan entre distintos módulos (procedimientos, funciones o subrutinas) en contraposición a aquellas optimizaciones efectuadas a nivel de sentencia, a nivel de arquitectura específica o a nivel de bucle.

Entre estas técnicas se destacarán las siguientes:

- Reemplazar llamadas a una función o subrutina por el código fuente (*procedure inlining).*
- Propagación de constantes.
- Eliminación de funciones innecesarias.
- Eliminación de variables innecesarias.
- Hacer precarga de los datos a memoria (*prefetch*).

8.6.1 PROCEDURE INLINING

8.6.1.1 INTRODUCCIÓN

Esta técnica de optimización consiste en reemplazar llamadas a una función o subrutina por su propio código fuente.

Los candidatos para *inlining* son módulos que:

- Son "pequeños".
- Son llamados a menudo.
- No consumen mucho tiempo por llamada.

8.6.1.2 VENTAJAS

Se obtiene las siguientes ventajas:

- Incrementar las oportunidades para realizar optimizaciones.
- Más oportunidades de intercambio de bucles.

- Incrementa las posibilidades de auto-paralelización.
- Reduce el sobrecoste de llamada (normalmente este efecto es mínimo).

8.6.1.3 **EJEMPLO**

```
Do i = 1, N
        A( i) = B( i) * 2. 0
        Call Ejemplo( A( i), C( i))
End Do
Subroutine Ejemplo(X, Y)
Y = 1 + X * (1.0 + X* 0.5)
```

Esto se puede sustituir por:

```
Do i = 1, N
        A( i) = B( i) * 2. 0
        C( i) = 1 + A( i) * (1.0 + A( i)* 0.5)
End Do
```

De esta forma se ha evitado tener que llamar N veces a la función Ejemplo dentro de un bucle.

8.6.1.4 **EJEMPLO 2**

```
DIMENSION A (100000,10000), B (20000,100000), C (50000,50000)
DO I= 1,10000000
        CALL RUTINA (A, B, C)
ENDDO
```

En este ejemplo estamos realizando una llamada en un bucle que realiza 10 millones de pasos.

Supongamos un formato de 64 bits (esto es 8 bytes para almacenar cada dato). En cada llamada estamos pasando matrices de tamaño:

A: 100000x10000 * 8 bytes = 7.45 GB

B: 20000x100000 * 8 bytes = 14.9GB

C: 50000x50000 * 8 bytes =18.62 GB

En total se está utilizando nada menos que 42 GB de datos saturando la llamada a la rutina.

Es por ello que se debe evitar llamar a RUTINA dentro del bucle a toda costa.

8.6.2 PROPAGACIÓN DE CONSTANTES

En muchas situaciones, se llaman a funciones o procedimientos con variables como argumentos que son constantes.

Además, esas variables pueden ser incluso que se estén pasando por referencia en cuyo caso sólo hay una referencia en todo el programa o que se esté pasando por parámetro, en cuyo caso se hace una copia de la variable.

En general, la propagación de constantes nos permite eliminar variables innecesarias.

El principio básico de la propagación de constantes viene precedido por el siguiente ejemplo:

a=b+5

c=a

e=f+c

Como se puede apreciar, este código se podría simplificar por:

a=b+5

e=f+a

E incluso en el caso de no necesitarse luego la variable “a” podría simplificarse por:

e=f+b+5

8.6.2.1 VENTAJAS

- Reducción del número de variables en el programa que permite al compilador realizar otro tipo de optimizaciones.

- Eliminación de las decisiones que debe tomar el compilador sobre la variable, lo cual le va a permitir realizar optimizaciones que sino no llevaría a cabo por seguridad.

8.6.2.2 **EJEMPLO**

```
a=10

C=0

B=a*2

Call Ejemplo(B,C)

Subroutine Ejemplo(B,C)

C=B*5+B

End
```

En este caso, estamos llamando a la subrutina Ejemplo con una variable B que realmente es una constante.

Para ello, se debería modificar por:

```
a=10

C=0

b=20

Call Ejemplo(C)

Subroutine Ejemplo(C)

C=b*5+b

End
```

Aunque este ejemplo es muy sencillo y realmente no hace nada, nos debemos imaginar el caso complejo en el cual se esté utilizando punteros u otras técnicas que impiden optimizaciones posteriores.

8.6.3 ELIMINACIÓN DE FUNCIONES INNECESARIAS

8.6.3.1 INTRODUCCIÓN

La eliminación de funciones innecesarias dentro del código permite al compilador realizar asunciones que no podría hacer de otras formas.

En general, cuando un código viene escrito de hace muchos años y ha pasado por sucesivas manos de programadores, hay trozos o partes que aunque permanecen “por si acaso” no suelen ser nunca invocados.

Una revisión con un buen programa de análisis, permite localizar estas funciones que no se utilizan nunca.

8.6.3.2 VENTAJAS

- Reducción del tamaño del código.
- Optimización global del código.

8.6.3.3 EJEMPLO

a=0

… “a” permanece invariante durante todo el código…

IF (a= =1) THEN CALL VETE_CAMINO 1

ELSE CALL_VETE_CAMINO_0

Esto se debe sustituir por la llamada directa:

CALL VETE_CAMINO_0

8.6.4 ELIMINACIÓN DE VARIABLES INNECESARIAS

La eliminación de variables innecesarias dentro del código permite al compilador realizar asunciones que no podría hacer de otras formas.

Ello suele ser debido a asignaciones de variables a otras variables cuando realmente se pudiera utilizar la primera variable.

8.6.4.1 **VENTAJAS**

- Reducción del tamaño del código.
- Explotación del *pipeline.*
- Mejora de la caché.

8.6.4.2 **EJEMPLO**

Sea el siguiente ejemplo donde se asume que “a” no se va a utilizar más en el código:

x = z + y

a = x

x = 2 * a

Se pasaría a sustituir por:

b = z + y

a = b

x = 2 * b

Para simplificarlo por:

b = z + y

x = 2 * b

E incluso por:

x =2 * z + y

8.7 ACCESO A MEMORIA: CACHÉ, MEMORIA PRINCIPAL Y E/S

El acceso a la jerarquía de memoria en las mejores condiciones suele ser un punto clave a la hora de conseguir el mejor rendimiento de un código.

Se debe hacer especial hincapié en este apartado puesto que se puede optimizar el código para minimizar el impacto de un mal direccionamiento a memoria.

En general, es fundamental conocer cual es el hardware que tenemos por debajo, pero en el caso del acceso a memoria es fundamental.

Es por ello, que si no se conoce, se debe preguntar al administrador la siguiente información:

- Tamaño de memoria que se puede utilizar. Aunque un sistema tenga mucha memoria, un administrador puede restringir su uso a un tamaño X.
- Tipo de memoria.
- Tipo de caché de primer, segundo y tercer nivel.
- Tamaño de cachés.
- Tipo de TLB.
- Tamaño de página.

Con ello podremos realizar muchas técnicas de mejora de las prestaciones.

En este TFC se verán las siguientes técnicas:

- Acceso en el mismo orden que el almacenamiento.
- División por bloques.
- Inserción de variables.
- Aumentar el tamaño de página.
- Realizar precarga de los datos.
- Mejorar la Entrada / Salida.

8.7.1 ACCESO EN EL MISMO ORDEN QUE EL ALMACENAMIENTO

Como se vio en las técnicas de optimización de bucles, acceso secuenciales de los datos aseguran que una vez que una línea es almacenada en la caché, todos los datos de la línea de la caché podrán ser referenciados sin dar una pérdida de caché.

El acceso secuencial a los datos permite también poder realizar un *prefetch* sobre arquitecturas que permitan *prefetch* por hardware para adelantar datos de memoria.

Es por ello, que una buena codificación de Fortran y C nos asegurará que los datos son accedidos de la misma forma que fueron almacenados.

Mantener un orden de *stride* 1 de acceso a los datos suele ser la técnica más básica para mantener un buen acceso a caché, aunque no siempre sea posible.

8.7.2 DIVISIÓN POR BLOQUES

En caso de no poder realizarse un *stride*, lo mejor será utilizar la técnica de división de bloques que se vio en la optimización de bucles.

La técnica de división de bloques nos va a permitir introducir los datos en la caché. Se basa en el concepto de romper la estructura de los datos que son demasiado grandes para que quepan en caché.

8.7.3 INSERCIÓN DE VARIABLES (COMMON BLOCK ARRAY PADDING)

8.7.3.1 INTRODUCCIÓN

Es muy común programar las dimensiones de las matrices o vectores en múltiplos de 2: 512, 1024, 2048, etc.

Este hábito suele conllevar a un mal acceso a la caché al solaparse en la misma línea de caché elementos del mismo *array*.

Los computadores tratan a los *array* de dos dimensiones como si fueran vectores. Se almacena una fila detrás de otra o una columna detrás de otra dependiendo como se ha visto anteriormente si se está programando en C o en Fortran.

El problema surge cuando las dimensiones tienen relación directa con múltiplos de 2 como pueden ser 2048, 4096, 8192 al guardar *REAL*8* en Fortran o variables *double* en C.

Aunque ninguna dimensión de los vectores sea múltiple de 2048, cada 4 filas (para C) o 4 columnas (para Fortran) es igual a 2048.

Cuando el *array* es almacenado en la caché, el primer elemento de la primera fila (o columna) mapeará en la misma posición del primer elemento de la quinta fila (o columna). Consecuentemente, todos los elementos de la primera y quinta fila (o columna) darán fallos de caché. Del mismo modo todos los elementos de la segunda y sexta fila (o columna) presentarán el mismo problema. Y así con el resto…

El *array* está siendo cacheado en trozos de 4 filas o columnas por lo que, aunque tengamos una caché grande, sólo estamos utilizando una parte muy pequeña. Todo ello conlleva a muchos fallos de caché resultando en un pésimo rendimiento.

Para solucionar este problema, se debe dimensionar los *array* introduciendo nuevos elementos.

Para Fortran, esto significa tener *array* de N+1 filas. Cuando se almacena en memoria, los elementos serán desde A(1,1) hasta A(513,1), A(1,2) hasta A(513,2) y así sucesivamente. Entonces 513 no es múltiplo de 2048 por lo que los primeros elementos de las columnas nunca caerán en el mismo espacio de direcciones de caché.

En C se necesitaría fijar las columnas a N+1 para que no haya múltiples filas que sean igual a 2048.

8.7.3.2 **VENTAJAS**

La ventaja fundamental de esta técnica consiste en el aumento del uso de la caché.

Adicionalmente se mejorar el software *pipeline*.

8.7.3.3 **EJEMPLO**

Sea el siguiente ejemplo:

```
COMMON /SHARED/A (1024,1024), B(1024,1024), C(1024,1024)

DO j = 1, 1024

    DO i = 1, 1024

        A (i,j) = A(i,j) + B(i,j)*C(i,j)
```

```
        END DO
END DO
```

Suponiendo una caché de 32 KB se tiene que:

Dir[C(1,1)] = Dir[B(1,1)] + 1024*1024*4

Entonces la posición de los elementos de C y de B va a coincidir en la misma línea de caché por cada iteración.

C(1,1) = B(1,1) ya que (1024*1024*4) mod 32KB = 0

B(1,1)…B(8,1) y C(1,1)…C(8,1)
B(9,1)…B(16,1) y C(9,1)…C(16,1)
…
…
…
B(1017,1)…B(1024,1) y C(1017,1)…C(1024,1)

Ahora se va a introducir "*padding*" entre ambas matrices:

```
COMMON /SHARED/A(1024,1024),PAD1(129), B(1024,1024),PAD2(129),
C(1024,1024)
DO j = 1, 1024
        DO i = 1, 1024
                A(i,j) = A(i,j) + B(i,j)*C(i,j)
        END DO
END DO
```

Entonces ahora los elementos de C y B en cada iteración no coinciden en la misma dirección de caché:

C(1,1)= B(1,1) + 1024*1024*4 + 129*4 = B(129,1) mod 32KB

B(1,1)…B(8,1)
B(9,1)…B(16,1)
…
B(129,1)…B(136,1) y C(1,1)…C(8,1)
…
…

De esta forma se pueden realizar 128 iteraciones consecutivas con datos que están en caché, mejorando además el software *pipeline*.

8.7.4 AUMENTAR EL TAMAÑO DE PÁGINA

8.7.4.1 INTRODUCCIÓN

Si el sistema operativo lo permite, utilizar páginas de memoria de mayor tamaño permite obtener mejoras de prestaciones en aquellos códigos que sean intensivos del uso de memoria.

Generalmente las tablas de página suelen ser de tamaño 16k, 256k, 1024k, 4096k, 16384 KB e incluso puede llegar a fijarse valores mayores.

De esta forma se evitará estar haciendo mucha paginación ya que se traerá muchos más datos de una vez de memoria.

8.7.5 ALINEAMIENTO DE LOS DATOS

Para un rendimiento óptimo, hay que asegurar que los datos sean alineados de una forma natural.

Un límite natural es una dirección de memoria que es múltiplo del tamaño del dato.

Por ejemplo, un *REAL (KIND=8*) se alinea de una forma natural si su dirección de inicio es múltiplo de 8.

La mayoría de los compiladores tratan de alinear los datos siempre que pueden. No obstante los *EQUIVALENCE statements* pueden causar que los datos se conviertan en no alineados.

Es por ello que los compiladores pueden tener problemas alineando los datos con *common blocks* y otras estructuras.

En general se deben seguir las siguientes reglas que ayudarán a tener un alineamiento natural de los datos:

1. Especificar cuidadosamente el orden y tamaño en la declaración de los datos de cada *common block*, tipo derivado y estructura record.

2. Empezar con el item numérico de tamaño más grande, seguido del más pequeño y seguido luego por datos no numéricos (caracteres).

3. Evitar un tamaño de *array* que sea múltiplo del tamaño de la caché.

4. Poner especial atención sobre los mensajes de aviso del compilador sobre datos no alineados.

5. En C/C++, suele ser mejor cargar un *array* grande y entonces utilizar un puntero para acceder a secciones sobre él, para evitar sobrescribir de esta forma líneas de caché.

6. Agrupar datos usados a la vez como en el siguiente ejemplo:

```
d=0.0
do i=1,n
    j=ind(i)
    d=d+sqrt(x(j)*x(j)+y(j)*y(j)+z(j)*z(j))
enndo
```

Se sustituye por:

```
d=0.0
do i=1,n
    j=ind(i)
    d=d+sqrt(r(1, j)*r(1, j)+r(2, j)*r(2, j)+r(3, j)*r(3, j))
enndo
```

8.7.6 HACER PRECARGA DE LOS DATOS (PREFETCH)

8.7.6.1 INTRODUCCIÓN

Hacer una precarga de los datos consiste en insertar instrucciones dentro del código (bien manualmente o bien vía directivas de compilador) para solicitar variables por adelantado a memoria y de esta forma minimizar la latencia de acceso a memoria mientras se están realizando otros cálculos.

8.7.6.2 VENTAJAS

- Facilita las prestaciones superescalares y de ejecución fuera de orden del procesador.

8.7.6.3 DESVENTAJAS

- Cuando se utiliza mal, añade ciclos extras que pueden repercutir negativamente en las prestaciones.

8.7.6.4 EJEMPLO

Sea el siguiente código:

```
for (i=0; i<n; i++) {
        a += b[i];
}
```

Supongamos que b es un *array* de tipo double, y que b[i+16] es una línea de la caché de datos L2 de tamaño 128 bytes.

Cada iteración del bucle tarda 2 ciclos en realizarse (*prefetch* más 1 *load/flop*) mientras que una pérdida de caché L2 tarda alrededor de 60 ciclos o más en pedir a memoria el dato.

Si se realiza un *prefetch* se adelanta los datos 32 ciclos (16 veces 2 ciclos), con lo cual se está escondiendo aproximadamente la mitad (32/60) de la latencia.

```
for (i=0; i<n; i++) {
```

```
        prefetch b[i+16];

        a += b[i];

}
```

De hecho, una mejor opción con menos sobre coste sería la siguiente:

```
for (i=0; i<n; i++) {

        if ((i%16) ==0) prefetch b[i+16];

        a += b[i];

}
```

Sin embargo, el sobrecoste de la sentencia "if" no compensa el ahorro de instrucciones de precarga. Es por ello que la mejor opción sería en conjuntarlo con un *loop unrolling* de la siguiente forma:

```
for (i=0; i<n; i+2) {

        prefetch b[i+16];

        a += b[i];

        a += b[i+1];

}
```

Con ello tenemos 8 *prefetch* por línea de caché solamente y realizando un *unroll* de 16 eliminamos totalmente los precargas redundantes. Ahora el compilador puede encargarse de esos detalles y no sólo de una línea de caché por adelantado, sino que puede encargarse de varias de ellas.

De esta forma, toda la latencia queda minimizada.

8.7.7 ENTRADA Y SALIDA EFICIENTE

Las siguientes guías permiten ejecutar una E/S eficiente en los códigos:

- Usar ficheros no formateados siempre y cuando sea posible. La E/S no formateada de datos numéricos es más eficiente y precisa que la formateada. Cuando se escribe en ficheros formateados, los datos son convertidos en caracteres para poder ser enviados a la salida, por lo tanto se puede enviar muchos menos datos en una sola operación de E/S y de hecho pueden perder precisión si los datos son leídos de vuelta en formato binario. Aunque los ficheros con formato son más fáciles de

portar a otros sistemas, la mayoría de los compiladores soportan conversión entre formatos nativos.

- Escribir el *array* completo en vez de elementos individuales. Para eliminar una sobrecarga innecesaria, realizar la escritura de un *array* completa de una vez es mejor que la de elementos individuales múltiples veces, ya que cada item en una lista de E/S genera su propia secuencia de llamada. Para ello utilizar la semántica Fortran 90/95 para tal efecto en vez de implementar con un bucle llamadas individuales.

- Escribir los datos en el orden natural de almacenamiento. Si no se puede usar el orden natural, puede ser más eficiente reordenar los datos en memoria antes de ejecutar la E/S.

- Usar *buffered I/O* siempre que sea posible. El defecto para Fortran es usar escrituras en disco sin buffer. Esto significa que los registros son escritos a disco inmediatamente, en vez de ir acumulándolo en un buffer parcial para que, cuando esté lleno, ejecute la instrucción de E/S a disco. Cuando se utiliza escritura en buffer se hace un uso mucho más eficiente de la E/S a disco escribiendo bloques más grandes de datos. La única desventaja es que un fallo de escritura puede causar pérdidas de registros puesto que se suponen que fueron escritos a disco cuando realmente no se hizo.

8.8 MISCELÁNEAS

Dentro de este grupo se muestra una serie extensa de otras técnicas que no se pueden agrupar en los apartados vistos anteriormente.

8.8.1 UTILIZACIÓN DE LIBRERÍAS ESTÁTICAS

Se conoce con el nombre de librería a la recolección de ficheros objetos bajo un mismo fichero.

Las librerías son usadas para proteger a las compañías que desarrollan software al permitir dar los ficheros objetos de una forma agrupada sin dar el código fuente, además de simplificar el uso de dichas librerías sin que el usuario conozca los detalles de la compilación de las mismas.

La creación de librerías se hace de una forma muy sencilla en todos los UNIX mediante el comando ar.

Existen 2 tipos de librerías:

- *Shared: Dynamic Shared Objects, DSO's (*. so).*
    - Comparten el segmento de código (no datos) de todos los usuarios.
    - Ahorran memoria (1 sola copia).
    - Son algo más lentas de ejecución puesto que se crea un nuevo segmento de datos al iniciarse el programa.
- *Non_ shared: static objects (*. a)*
    - Más rápidas de ejecución.
    - Mayor ocupación de memoria.
    - No todas las librerías existen en este formato.
    - No se instalan por defecto.

Por lo tanto es mejor utilizar las librerías estáticas más que utilizar librerías dinámicas si nuestro objetivo es que el código se ejecute de la forma más rápida posible. La única contrapartida es que ocuparán mayor espacio tanto en memoria como en tamaño del ejecutable creado.

8.8.2 ELECCIÓN DE LAS OPERACIONES

Esta técnica consiste en sustituir operaciones aritméticas que son costosas en ciclos de reloj por otras iguales que no lo son.

La división, módulo o raíz cuadrada no son traducidas directamente en instrucciones hardware ya que el compilador combina operaciones FMA, FRCPA, FRSQRA para realizar iteraciones Newton para ejecutarlas.

Se denomina latencia de instrucción al número de ciclos de reloj que son necesarios para ejecutar una instrucción después que los datos de entrada estén disponibles.

Se denomina rendimiento de instrucción al número de ciclos de reloj que el procesador tiene que esperar a empezar la ejecución en una misma operación. Este valor es siempre menor o igual a la latencia de instrucción.

Teniendo en cuenta estos valores, el impacto sobre la elección de un algoritmo es directo.

En las siguientes tablas se muestra ejemplos con la latencia de instrucción en ejecutar diferentes operaciones en un procesador Intel Itanium 2:

Latencia FP	y = a+y	y = a*y	y = a+b*y	y = b+a/y	y = a/sqrt(y)	y = sqrt(y)	y = y/sqrt(y)
Simple	4	4	4	28	36	43	36
Doble	4	4	4	32	37	55	37

Latencia Int	i = i + c	i = a*i	i = a+b*i	i = b+ a / i	i = b + a % i
Simple	1	15	16	37	42
Doble	1	15	16	56	61

La diferencia entre una multiplicación y una división es clara, 28 ciclos frente a 4.

Por ello se deben sustituir, siempre que se pueda, operaciones de división por operaciones de multiplicación.

A continuación se muestra un ejemplo:

```
subroutine CambioAMultiplicacion (a)

dimension a(1000)

do i=1,1000

        a(i) = 0.0

end do

b = 2.0

do i=1,1000

        a(i) = a(i) / b

end do

end
```

Se observa que en el segundo bucle se está realizando una división entre los elementos del vector por una constante.

```
subroutine CambioAMultiplicacion(a)

dimension a(1000)

do i=1,1000

        a(i) = 0.0

end do

b = 1.0 / 2.0

do i=1,1000

        a(i) = a(i) * b

end do

end
```

Se propone cambiar la operación de división por 2, por una multiplicación por 0,5.

Con ello estamos ahorrando multitud de ciclos de reloj y este bucle puede tomar hasta 7 veces menos en ejecutarse.

Otro ejemplo muy sencillo sería:

```
subroutine CambioASuma (a,b)

dimension a(1000), b(1000)

do i=1,1000

        a(i) = 0.0

end do

do i=1,1000

        b(i) = a(i) * 2

end do

end
```

La multiplicación de enteros es más costosa que la suma de enteros.

Por ello, cambiamos la multiplicación por la suma:

```
subroutine CambioASuma (a,b)

dimension a(1000),b(1000)

do i=1,1000

        a(i) = 0.0

end do

do i=1,1000

        b(i) = a(i) +a(i)

end do

end
```

8.8.3 FACTOR COMÚN

La técnica de factor común consiste en sacar fuera de los bucles aquellas operaciones que son costosas de realizar (por ejemplo: senos, cosenos, raíces cuadradas, divisiones, etc.) y que son independientes de los índices del bucle o independientes de posibles contadores dentro de los bucles. Estas operaciones se pueden realizar una única vez y dejarlas como constantes para ser introducidas en los bucles. De esta forma evitamos el hecho que se realicen n veces siendo n el número de iteraciones del bucle.

A continuación se muestra un ejemplo de factor común:

```
DO I= 1,1000

        DO J= 1,3000

        A( I, J) = SIN( 2* PI/ I)* B( I, J) + COS( PI/ 3) * B( I, J)

        ENDDO

ENDDO

CP = COS( PI/ 3)

DO I= 1,1000

        SP = SIN( 2* PI/ I)

        DO J= 1,3000
```

```
            A( I, J) = SP * B( I, J) + CP * B( I, J)
        ENDDO
ENDDO
```

Como se puede comprobar en este ejemplo, la operación:

A(I, J) = SIN(2* PI/ I)* B(I, J) + COS(PI/ 3) * B(I, J)

Se estaba realizando un total de:

- Divisiones: 2 operaciones: PI / I y PI / 3.
- Multiplicaciones: 3 operaciones: 2* PI / I, SIN(2* PI/ I)* B(I, J) y COS(PI/ 3) * B(I, J).
- Senos: 1 operación: SIN(2* PI/ I).
- Cosenos: 1 operación: COS(PI/ 3).
- Sumas: 1 operación: SIN(2* PI/ I)* B(I, J) + COS(PI/ 3) * B(I, J).

El número de iteraciones es 1000*3000= 3 millones de veces con lo cual tenemos que el total de operaciones es:

- Divisiones: 6 millones.
- Multiplicaciones: 9 millones.
- Senos: 3 millones.
- Cosenos: 3 millones.
- Sumas: 3 millones.

Primero se ha sacado como constante la operación COS (PI/3) puesto que es independiente de los índices del bucle.

Posteriormente, aquellas operaciones que dependían del índice i pero no del índice j, han sido sacadas fuera del bucle interno:

SP = SIN(2* PI/ I)

Con ello el número de operaciones a realizar es:

Fuera de los bucles:

- Cosenos: 1 operación: COS(PI/ 3).
- Divisiones: 1 operación: PI/3.

Dentro del bucle I el cual se ejecuta 1000 veces:

- Senos: 1 operación: SIN(2* PI/ I) -> 1000 operaciones.
- Divisiones: 1 operación: PI/ I -> 1000 operaciones.
- Multiplicación: 1 operación: 2* PI –> 1000 operaciones.

Dentro del bucle j el cual se ejecuta 3000 veces:

- Multiplicación: 2 operaciones –> 6 millones de operaciones.
- Sumas: 1 operación -> 3 millones de operaciones.

Con esta optimización se ha pasado a tener que realizar:

- Divisiones: De 6 millones a 1001.
- Multiplicaciones: De 9 millones a 6 millones mil.
- Senos: De 3 millones a 1000.
- Cosenos: De 3 millones a 1.
- Sumas: Permanece igual.

Este ejemplo refleja cómo cambiando el código con una operación básica, conseguimos aumentar las prestaciones en gran medida.

Sin embargo no siempre se puede realizar operaciones de factor común:

```
do i= 1,1000
        do j= 1,1000
                a( j, i) = sqrt( i* j)
        end do
end do
```

En el ejemplo anterior, la operación sqrt(i* j) es dependiente de los índices i y j de los índices, por lo tanto es imposible hacer una operación de factor común.

8.8.4 DEPENDENCIA DE DATOS Y PARALELISMO DE INSTRUCCIONES

Como añadido a la latencia y rendimiento de instrucción, la dependencia de datos afecta a la posibilidad del procesador de ejecutar instrucciones simultáneamente.

Cuando un algoritmo se estructura de tal forma que se pueden ejecutar sus instrucciones simultáneamente por el procesador, el algoritmo tomará menos tiempo en ser completado.

El Pentium 4 de Intel puede ejecutar 6 instrucciones por ciclo de reloj como ejemplo, aunque normalmente no ejecuta este máximo debido a la dependencia de datos.

Imaginemos que tenemos que realizar la siguiente operación:

a=b*c

e=f*g

Suponiendo que se tarda 8 ciclos de reloj de latencia de instrucción y 4 de rendimiento de instrucción, estas 2 operaciones se podrán realizar en paralelo debido a que no hay dependencia de datos, con lo que tomará 12 ciclos de reloj en completarse.

Sin embargo, si tomamos en consideración la siguiente operación:

a=b*c

d=a*e

Para realizar la segunda multiplicación es necesario que se haya acabado de realizar la primera multiplicación puesto que se necesita saber el resultado del valor de “a”.

En este caso, las 2 operaciones tomarían 20 ciclos de reloj.

Las dependencias de datos no son siempre posibles de romper pero se debe tratar de hacer un esfuerzo.

A continuación se analiza el siguiente ejemplo:

```
a = 0;

for (i=0; i<10000; i++)

   a += buf[i];
```

El incremento de la variable i y la suma de a +buf[i] aparentemente parece que se pueden hacer al mismo tiempo por no tener dependencias. Pero esto no es así puesto

que existe dependencia entre iteraciones puesto que el valor de la iteración i depende de la iteración i+1.

Por ello, en conjunción con la técnica de *unroll* de bucles vista anteriormente, se desenrollará este bucle en 4 para que se puedan realizar más operaciones por ciclo debido a que habrá pocas dependencias de datos.

```
a = b = c = d = 0;

for (i=0; i<10000; i+=4)

{

   a += buf[i];

   b += buf[i+1];

   c += buf[i+2];

   d += buf[i+3];

}

a = a + b + c + d;
```

Aunque cada bucle ahora ejecuta más instrucciones que antes, el número total de iteraciones puede ser reducido en cuatro, por lo que este bucle se ejecutará más rápido. Hoy en día los modernos procesadores han introducido instrucciones de vectorización para realizar este tipo de optimizaciones, como la instrucción SSE2 del procesador Pentium 4 y en adelante de Intel.

En líneas generales se puede decir que se ha conseguido un resultado aceptable cuando el procesador puede ejecutar 4 o más operaciones simultáneamente.

Hay opciones de compilación en la mayoría de los compiladores que nos van a permitir sacar un reporte de estos parámetros.

8.8.5 FUNCIONES INTRÍNSECAS

Esta técnica consiste en substituir llamadas a funciones intrínsecas tales como sin, cos, atan, log, sqrt, etc. por una llamada "vectorizada" que realice los cálculos más rápidamente.

En general requiere de un mínimo de longitud (~10) para compensar los costes de inicialización de la llamada.

Se puede implementar directamente o esperar que el compilador lo implemente de forma automática.

8.8.5.1 **EJEMPLO**

Tengamos la siguiente función:

```
SUBROUTINE vintr(n, x, y)

IMPLICIT NONE

INTEGER i, n

REAL x(n), y(n)

DO i = 1, n

        x(i) = i

        y(i) = sin(x(i))

END DO

RETURN

END
```

Puede ser transformado por el siguiente código:

```
        SUBROUTINE vintr(n, x, y)

        IMPLICIT NONE

        INTEGER*4 n

        REAL*4 x(n)

        REAL*4 y(n)

C **** Variables temporales ****

C

        INTEGER*4 i

C

C **** statements ****
```

C

```
      DO i = 1, n, 1
            x(i) = REAL(i)
      END DO
      CALL vsinf$(x(1),y(1),%val(n),%val(1_8),%val(1_8))
      RETURN
      END ! vintr
```

Esta llamada sustituye al bucle anterior y se ejecutará mucho más rápido si el número de iteraciones n es suficientemente grande (mayor que 10).

8.8.6 ARITMÉTICA Y PRECISIÓN ESTÁNDAR

Para muchas aplicaciones, el control del error numérico es fundamental, y el programador tiene mucho cuidado con las operaciones numéricas para que la precisión finita de los registros del computador no introduzca y propaguen errores. De esta forma surgió el estándar IEEE 754 para preservar la precisión sobre operaciones aritméticas sobre flotantes por encima de la velocidad de ejecución si fuera necesario y la conformidad con esta norma es un elemento clave de numerosos programas numéricamente sensitivos, especialmente de aquellos códigos que deben producir respuestas idénticas en diferentes entornos.

Muchos otros programas y programadores son menos rígidos acerca de preservar cada bit de precisión y están más interesados en obtener respuestas lo más pronto posible a pesar de perder información en redondeos.

Dependiendo de estas condiciones, el programador puede decirle al compilador que grado de flexibilidad o rigidez debe acometer con las operaciones matemáticas.

8.8.7 CONFORMIDAD IEEE

Los procesadores actuales suelen tener instrucciones hardware para ejecutar operaciones flotantes especializadas en un tiempo muy rápido. Sin embargo estas instrucciones hardware no suelen cumplir el estándar IEEE. Aunque la inexactitud sea pequeña (no más de dos dígitos por ejemplo) un programa que usa estas instrucciones no cumple estrictamente el estándar IEEE y podría generar un resultado diferente del mismo programa cumpliendo esta norma.

Los compiladores actuales sin embargo, son capaces de controlar la adhesión al estándar IEEE mediante opciones de compilación con lo cual se puede controlar que el código generado se adhiere a dicho estándar para operaciones aritméticas.

También, algunas de las técnicas de optimización básicas que se están viendo en el trabajo, pueden producir resultados que no se adecuan al estándar IEEE.

Se va a mostrar a continuación un ejemplo de una optimización estándar que no cumple el estándar IEEE. Esta optimización es implementada en los compiladores actuales.

```
do i = 1, n

        b(i) = a(i)/c

enddo
```

Optimizando el bucle queda de la siguiente forma:

```
tmp = 1.0/c

do i = 1, n

        b(i) = a(i)*tmp

enddo
```

Esta operación no está permitida por el estándar IEEE porque multiplicar por el recíproco en lugar de una división puede producir resultados que difieren en la parte menos significativa.

Otro ejemplo lo tendríamos en la operación x / x el cual puede sustituirse por 1.0 sin realizar la operación, pero que el estándar IEEE lo prohíbe.

Afortunadamente los compiladores permiten manejar esto de diferentes formas: obligar a cumplir completamente el estándar, permitir usar instrucciones hardware que sean débilmente inexactas o permitir al compilador usar cualquier operación que sea algebraicamente válida aunque haya perdidas de redondeo.

8.8.8 CONTROL DEL REDONDEO

El orden en las cuales las operaciones aritméticas son codificadas en el programa dicta el orden en el cual son ejecutadas. El siguiente ejemplo muestra que la variable sum es calculada sumando primeramente a(1) a sum y al resultado se le suma a(2) y así hasta que a(n) ha sido sumado a los resultados.

```
sum = 0.0
```

```
do i = 1, n
        sum = sum + a(i)
enddo
```

En muchos casos la secuencia programada de operaciones no producirá necesariamente un resultado numéricamente válido.

Supongamos que el procesador tiene una unidad de suma flotante con *pipeline* con una latencia de 2 ciclos. Cuando el compilador reordena el código puede desenrollar el bucle de esta forma:

```
sum1 = 0.0
sum2 = 0.0
do i = 1, n-1, 2
        sum1 = sum1 + a(i)
sum2 = sum2 + a(i+1)
enddo
do i = i, n
        sum1 = sum1 + a(i)
enddo
sum = sum1 + sum2
```

El compilador desenrolla el bucle para poner 2 sumas en cada iteración. Esto reduce la sobrecarga del incremento de índice a la mitad y también el abastecimiento a pares a las unidades de suma flotante para mantener al sumador de 2 estados ocupado en cada ciclo.

El resultado algebraico de sum es idéntico en ambos casos pero debido a la naturaleza finita de los computadores, estos 2 bucles pueden producir resultados que difieren en sus bits finales, bits que son denominados error de redondeo.

Los compiladores actuales permiten especificar como de estricto debe acometer este tipo de operaciones: inhibir cualquier tipo de reordenado de bucles, permitir reordenado simples con expresiones, reordenados extensivos o permitir cualquier tipo de transformación algebraicamente válida.

Esta última opción le permite al compilador una total libertad de maniobra en operaciones como por ejemplo cambiar flotantes a enteros. Lógicamente se ve que este tipo de operaciones puede llevar a resultados de precisión incorrecta para un científico riguroso.

8.8.9 EXCESIVO ANÁLISIS RECURRENTE

Desde el punto de vista de la Programación Estructurada el análisis recurrente es un buen método de aplicación de la técnica "divide y vencerás" a la resolución de algoritmos complejos.

A pesar de ello, su implementación directa puede generar sobrecarga y dificultades para el compilador para optimizar y paralelizar el código.

Por lo tanto se tiene que estudiar si es mejor utilizar a veces un algoritmo recurrente para ganar eficiencia algorítmica o bien un método más directo que sin embargo permita al compilador optimizar mejor el algoritmo.

Un ejemplo de ello lo tenemos en el siguiente programa de Fortran que calcula el factorial de un número.

En pseudocódigo se tiene que el factorial de un número N es:

Si N <=0 entonces Factorial(N) = 1

Sino Factorial(N) = N*Factorial(N-1)

Pasando este pseudocódigo a Fortran tenemos que:

```
FUNCTION FACTORIAL (N)

INTEGER N

IF (N. LE. 1) THEN

      RETURN 1

ELSE

      RETURN (N*FACTORIAL(N-1))

ENDIF
```

Como podemos comprobar en el ejemplo anterior, desde el punto de vista de Programación, el código es una implementación directa del pseudocódigo pero no resulta lo más eficiente puesto que para paralelizar el código tendríamos que modificar el código completamente e incluso el compilador tampoco puede optimizar

prácticamente nada (aunque las últimas versiones de compiladores ponen mucho énfasis en tratar de optimizar las llamadas entre procedimientos o funciones)

Además en este ejemplo para el cálculo de un factorial grande estamos llenando la pila de ejecución con numerosas llamadas recurrentes.

Una implementación mejorada de la función factorial sería la siguiente

```
FUNCTION FACTORIAL (N)

INTEGER N,R

R=1

DO I= 1, N

        R = R*N

END DO

RETURN R
```

En este ejemplo el compilador o el programador, puede optimizar perfectamente el código haciendo por ejemplo software *pipelining* y otras técnicas más.

Además este código es directamente paralelizable simplemente añadiendo una directiva OpenMP:

```
FUNCTION FACTORIAL (N)

INTEGER N,R

R=1

#pragma omp parallel do private(I) reduction(*:R)

DO I= 1, N

        R = R*N

END DO

RETURN R
```

8.8.10 EXCESIVA CONCISIÓN

A la hora de escribir códigos, son muchos los programadores que van escribiendo sin pensar en que la implementación directa no es siempre el mejor de los caminos.

No por dejar el código lo más sencillo y explicativo posible significa también que el compilador va a poder realizar mejor las optimizaciones.

Un ejemplo de ello es cuando se utiliza excesiva concisión a la hora de escribir un algoritmo.

En este ejemplo de excesiva concisión se está escribiendo un código que tiene un bucle que contiene una estructura alternativa interior:

```
DO I= 1,1000000
    IF (MOD (i, 2). EQ. 1) THEN
        A (I)= A( I) + 1
    ELSE
        B (I) = B (I)*2
    ENDIF
ENDDO
```

Como se puede apreciar, el compilador tiene complicado realizar una buena optimización al haber introducido una sentencia alternativa que dependiendo de si el índice es par o impar hace una operación u otra.

Para este caso es mejor rescribir el código de la siguiente forma:

```
DO I= 1,1000000,2
    A( I)= A( I) + 1
ENDDO
DO I= 2,1000000,2
    B (I) = B(I)*2
ENDDO
```

De esta forma el compilador puede aplicar técnicas que se vieron anteriormente para optimizar bucles además de hacer mejor uso de la caché.

Otro ejemplo de excesiva concisión es emplear elementos que se salgan del estándar del lenguaje como por ejemplo utilizar sentencias "*goto*".

```
DO I= 1,3000000

...

      IF (A(I). EQ. 0) GOTO 200

...

ENDDO
```

El empleo de sentencias "*goto"* complican mucho la buena optimización del compilador además de complicar la lectura del código.

8.8.11 DEJAR AL COMPILADOR HACER SU TRABAJO

El compilador es el primer interfaz de acceso al procesador. Lenguajes de alto nivel como C o Fortran son traducidos en instrucciones de bajo nivel que son ejecutadas por el procesador con lo cual cuanto más optimizado esté un compilador más eficiente será la ejecución de las instrucciones.

A pesar de todo, se puede ayudar en gran medida al compilador a realizar su trabajo de una forma más eficiente. No obstante, la mejor guía es primero dejar al compilador hacer su trabajo y luego aplicar las técnicas anteriormente mencionadas.

Idealmente el compilador debería convertir el programa en un código que fuera la más rápido posible. Pero esto no es posible porque el compilador no tiene siempre la suficiente información para tomar las decisiones más óptimas.

A modo de ejemplo, si un bucle se ejecuta muchas veces, se ejecutará más rápido si el compilador desenrolla (*unrolls*) el bucle y ejecuta software *pipelining* sobre el código desenrollado (*unrolled*). Pero, si el bucle ejecuta pocas iteraciones, el tiempo gastado en elaborar código generado para realizar *pipelining* nunca es recuperado, por lo cual el bucle se ejecutará muy lentamente. Con algunos de los bucles DO de Fortran y muchos de C, el compilador no puede decidir cuantas veces se ejecutará este bucle ya que depende de operaciones anteriores que están por decidir en tiempo de ejecución. Por ello, la respuesta más directa sería decirle al compilador lo que tiene que hacer para realizar la optimización correcta. Para ello, es necesario comprender que tipos de optimizaciones suelen realizar los compiladores actuales y qué opciones de compilación *("compiler flags")* pueden ser habilitados o deshabilitados para realizar una buena optimización.

9 CONCLUSIONES Y DESARROLLOS FUTUROS

9.1 CONCLUSIONES

9.1.1 CONCLUSIONES SOBRE EL USO DE RECURSOS DE COMPUTACIÓN

En la actualidad, el mundo de la computación es uno de los campos en los cuales se está invirtiendo más dinero.

Según Intel, uno de cada cuatro procesadores que vende actualmente se destina para computación. La mayoría de los centros de computación destinan grandes partidas a la renovación de sistemas.

Sin embargo, en muchos casos, aunque se destina mucho dinero a la compra de equipos, es poco el presupuesto que se invierte en el desarrollo y optimización de software.

La mayoría de los centros de investigación actuales, están dotados de administradores que saben exclusivamente de sistemas, pero no saben ayudar a los científicos en la optimización de códigos.

Es por ello, que este trabajo fin de carrera ha querido servir de base de referencia a los administradores y científicos para poder empezar a plantearse un escenario en el cual no sea necesario adquirir cada cierto tiempo nuevo hardware, sino que sabiendo sacar el máximo rendimiento a lo que se tiene, se podría aumentar el tiempo de vida de los sistemas. De esta forma se podría invertir gran parte del presupuesto a la contratación de personal especialista en ayuda de optimización de códigos para los usuarios.

Aplicando muchas de las técnicas vistas en este trabajo y dada mi propia experiencia optimizando códigos para muchos de los centros de investigación de España, he visto que se puede llegar a optimizar un código para que se ejecute hasta 100 veces mejor. Aunque la mayoría de los códigos se consigue optimizar una media de mejora de 2 a 3 veces, es una ventaja considerable en prestaciones.

Que duda cabe, que si el presupuesto medio de un proyecto de computación en España, como puede ser la renovación de un sistema central en una Universidad, suele estar en torno a los trescientos mil euros, reservando una tercera parte de este presupuesto (cien

mil euros) a la contratación de un especialista en optimización de códigos podría llevar a la mejora de la usabilidad de los sistemas.

En mi propia experiencia vendiendo sistemas a la mayoría de los centros de investigación de España, muchos de dichos sistemas no se estaban utilizando ni tan siquiera al 30%. Otros sin embargo, estaban sobreexplotados y seguramente una mejora en los códigos hubiera permitido una mejora del rendimiento de los mismos.

9.1.2 CONCLUSIONES SOBRE LA METODOLOGÍA

En cuanto a la metodología desarrollada en este proyecto fin de carrera, las conclusiones obtenidas son las siguientes:

- Es necesario conocer cómo ha evolucionado la computación desde sus orígenes para comprender cómo se ha llegado en la actualidad al sistema que se está utilizando para ejecutar los códigos.

- Es imprescindible tener un conocimiento completo del hardware que se está utilizando. Se debe investigar y analizar exhaustivamente todos los parámetros del hardware que estamos utilizando ya que conocer tipo y tamaño de caché, tipo de memoria, tamaño de página, acceso a disco, velocidad de acceso, frecuencia de reloj, tipo de procesador, etc. es absolutamente necesario para sacar el máximo rendimiento a un código.

- La aplicación de una buena metodología de programación nos va a asegurar, por lo menos, que no se cometan errores contra el lenguaje que conlleven a un mal rendimiento de inicio.

- La elección del lenguaje a la hora de escribir un programa, va a influir mucho en el rendimiento. Un mismo programa escrito en dos lenguajes diferentes pueden dar enormes diferencias de rendimiento.

- Hay que dejar al compilador realizar su trabajo para luego decidir en que puntos hay que optimizar el código.

- Las herramientas de análisis de rendimiento son clave para localizar los cuellos de botella de un código.

- Los cuellos de botella más comunes en los códigos son debidos a un mal acceso a memoria y a un diseño incorrecto de los bucles.

- Es necesario el uso de herramientas de análisis que nos ayude a localizar donde están los cuellos de botella y posible soluciones.

Las desventajas que se aprecian a la hora de aplicar el método son las siguientes:

- La utilización de algunas de las técnicas vistas para mejorar el rendimiento en un código, puede oscurecer en algunas ocasiones la lectura del mismo, dificultando la comprensión del mismo a alguien que no sea el programador que lo diseñó.

- Al optimizar un código, muchas veces limitamos la portabilidad del mismo, al estar diseñado sobre un tamaño en particular de caché, de acceso a memoria, de arquitectura, etc.

- En muchas ocasiones, es difícil destinar tiempo a la tarea de optimización. Se requiere de conocimientos de programación y de arquitectura de computadores para poder realizar algunas de las técnicas vistas. Es por ello, que muchos investigadores no quieren realizarlas.

- El compilador es nuestro mejor aliado. Sin embargo, muchas veces necesita de nuestra ayuda para poder optimizar al máximo un código.

9.2 DESARROLLOS FUTUROS

Para futuros desarrollos en este área, se ha dejado la parte de paralelización de código ya que supone en si mismo un trabajo incluso más exhaustivo que el actual propuesto para optimización de código monoprocesador. Es por ello que se ha dejado fuera del alcance de este trabajo fin de carrera.

Adicionalmente, se ha dejado también la migración de códigos hacia GPGPU. Al igual que la paralelización, requiere unos conocimientos primero sobre optimización monoprocesador, luego sobre paralelización y por último habría que abordar el estudio sobre GPU.

10 BIBLIOGRAFÍA

[Campbell-Kelly, 1996]. Martin Campbell-Kelly. *Computer: a history of the information machine*. Basic Books, 1996.

[Cisneros, 2003]. Gerardo Cisneros. *SGI Altix Applications Development and Optimization*. Silicon Graphics, 2003.

[Cortesi and Fier, 1998] David Cortesi and Jeff Fier. *Origin2000 and Onyx2 Performance Tuning Optimization*. David Cortesi, based on the first edition by Jeff Fier. Silicon Graphics, 1998.

[De Bustos, 2002] Oscar de Bustos. *Curso de Optimización y Paralelización de Código sobre Origin 3000 en CSIC CTI*. Silicon Graphics, 2002.

[De Bustos, 2003] Oscar de Bustos. *Introducción a la Paralelización y Optimización de Código en CIEMAT*. Silicon Graphics, 2003.

[De Bustos, 2003] Oscar de Bustos. *Curso de Optimización y Paralelización de Código sobre Origin en CIEMAT*. Silicon Graphics, 2003.

[De Bustos, 2004] Oscar de Bustos. *SGI Altix Applications Development and Optimization Training en Puertos del Estado*. Silicon Graphics 2004.

[De Bustos, 2004] Oscar de Bustos. *Curso de Optimización y Paralelización de Código sobre Altix en la Universidad de Valencia*. Silicon Graphics, 2004.

[De Bustos, 2005] Oscar de Bustos. *Curso de Optimización y Paralelización de Código sobre Altix en la Universidad Politécnica de Valencia*. Silicon Graphics, 2005.

[De Bustos, 2006] Oscar de Bustos. *SGI Altix Applications Development and Optimization Training en EADS CASA*. Silicon Graphics 2006.

[Gerber, Bik, Smith and Tian, 2006] Richard Gerber, Aart J.C. Bik, Kevin B. Smith and Xinmin Tian. High-Performance Recipes for IA-32 Platforms. Intel Press, 2006.

[Koren, 2006] Gabi Koren. *Altix Application Performance Training*. Silicon Graphics, 2006.

[Nash, 1990] Stephen G. Nash. *A history of scientific computing*. ACM Press, 1990.

[Nvidia, 2008] Nvidia Corporation. NVIDIA *CUDA Compute Unified Device Ar1chitecture*. Nvidia, 2008.

[Patterson and Hennessy, 2005]. Patterson, D.A. y Hennessy, J.L. *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann, 2005

[Snyder, 2000] Kim Snyder. *SGI 2000 Series Applications Programming & Optimization*. Silicon Graphics, 2000.

[Trill, 2000] Albert Trill. *Curso de Introducción a la Programación sobre CrayOrigin 2000*. Silicon Graphics, 2000.

[Vogelsang, 2005] Reiner Vogelsang. *Altix Application Programming*. Silicon Graphics, 2005.

[Zacharov, 2001] Igor Zacharov. *Optimization and Parallelization Training. Silicon Graphics*, 2001.